Administrivia

Updated Midterm/Final grading policy

- if your final has a higher score, we will drop your midterm 1 score and weigh the final represent both exams

Administrivia

HW4

- Query optimization
- EXPLAIN query

Project 2

- Graph analysis using SQL
- Extra Credit: recursive WITH clauses

Both Due 11/21

Real-world Example of FDs

Airport Dataset

```
"id","ident","type","name","latitude_deg","longitude_deg","elevation_ft","continent","iso_country","iso_region","municipality","sche
6523,"00A","heliport","Total Rf Heliport",40.07080078125,-74.93360137939453,11,"NA","US","US-PA","Bensalem","no","00A",,"00A",,,
323361,"00AA","small_airport","Aero B Ranch Airport",38.704022,-101.473911,3435,"NA","US","US-KS","Leoti","no","00AA",,"00AA",,,
6524,"00AK","small_airport","Lowell Field",59.947733,-151.692524,450,"NA","US","US-AK","Anchor Point","no","00AK",,"00AK",,,
6525,"00AL","small_airport","Epps Airpark",34.86479949951172,-86.77030181884766,820,"NA","US","US-AL","Harvest","no","00AL",,"00AL",
6526,"00AR","closed","Newport Hospital & Clinic Heliport",35.6087,-91.254898,237,"NA","US","US-AR","Newport","no",,,,,,"00AR"
322127,"00AS","small_airport","Fulton Airport",34.9428028,-97.8180194,1100,"NA","US","US-OK","Alex","no","00AS",,"00AS",,,
6527,"00AZ","small_airport","Cordes Airport",34.305599212646484,-112.16500091552734,3810,"NA","US","US-AZ","Cordes","no","00AZ",,"00
6528,"00CA","small_airport","Goldstone (GTS) Airport",35.35474,-116.885329,3038,"NA","US","US-CA","Barstow","no","00CA",,"00CA",,,
324424,"00CL","small_airport","Williams Ag Airport",39.427188,-121.763427,87,"NA","US","US-CA","Biggs","no","00CL",,"00CL",,,
322658,"00CN","heliport","Kitchen Creek Helibase Heliport",32.7273736,-116.4597417,3350,"NA","US","US-CA","Pine Valley","no","00CN",
6529,"00CO","closed","Cass Field",40.622202,-104.344002,4830,"NA","US","US-CO","Briggsdale","no",,,,,,"00CO"
6531,"00FA","small_airport","Grass Patch Airport",28.64550018310547,-82.21900177001953,53,"NA","US","US-FL","Bushnell","no","00FA",,
6532,"00FD","closed","Ringhaver Heliport",28.8466,-82.345398,25,"NA","US","US-FL","Riverview","no",,,,,,"00FD"
6533,"00FL","small_airport","River Oak Airport",27.230899810791016,-80.96920013427734,35,"NA","US","US-FL","Okeechobee","no","00FL",
534,"00GA","small_airport","Lt World Airport",33.76750183105469,-84.06829833984375,700,"NA","US","US-GA","Lithonia","no","00GA",,"0
6535,"00GE","heliport","Caffrey Heliport",33.887982,-84.736983,957,"NA","US","US-GA","Hiram","no","00GE",,"00GE",,,
6536,"00HI","heliport","Kaupulehu Heliport",19.832881,-155.978347,43,"OC","US","US-HI","Kailua-Kona","no","00HI",,"00HI",,,
```

TPC-H Warehousing Benchmark

LineID, OrderID, PartID, SupplierID, Qty, Discount, Tax,
CustomerID, OrderTotalPrice, OrderDate,
AvailQty, SupplyCost,
PartName, Brand, Type
SupplierName, Address

Order → CustomerID, OrderTotalPrice, OrderDate
PartID,SupplierID → AvalQty, SupplyCost
PartID → PartName, Brand, Type
SupplierID → SupplierName, Address
LineID → OrderID, PartID, SupplierID, Qty, Discount, Tax

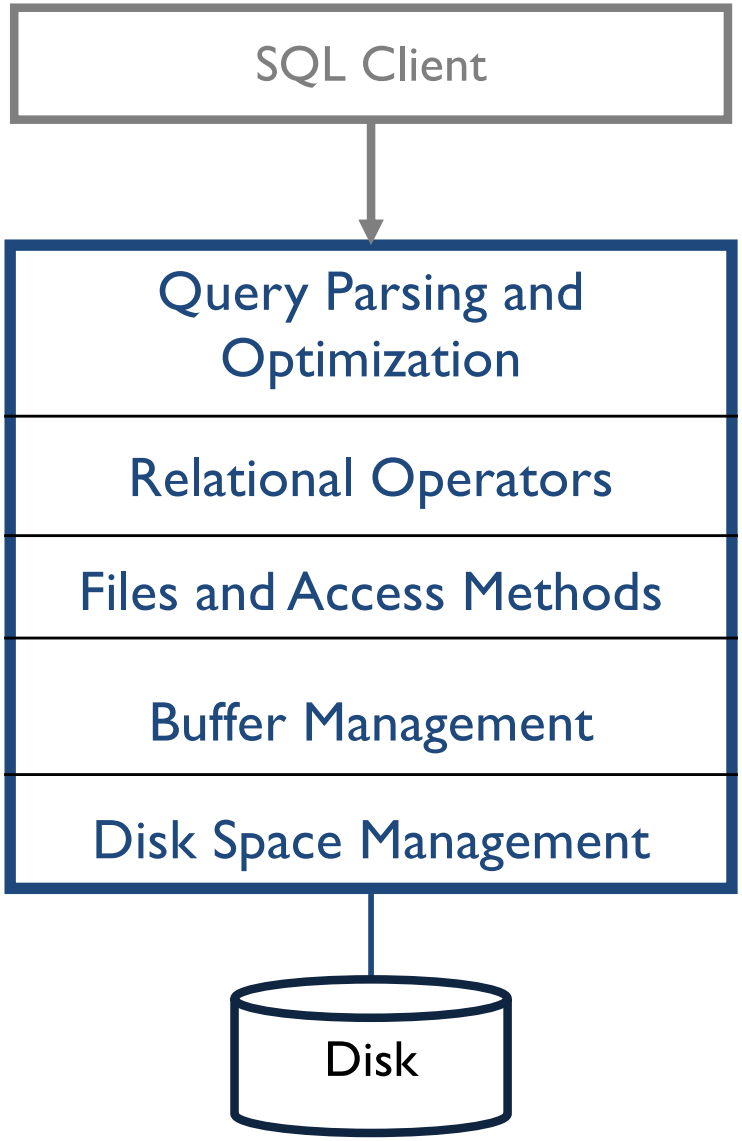
Disk, Storage, and Indexing

Eugene Wu

DBMS Overview

Each layer provides a simple abstraction to layers above it, and makes assumptions about layers below it.

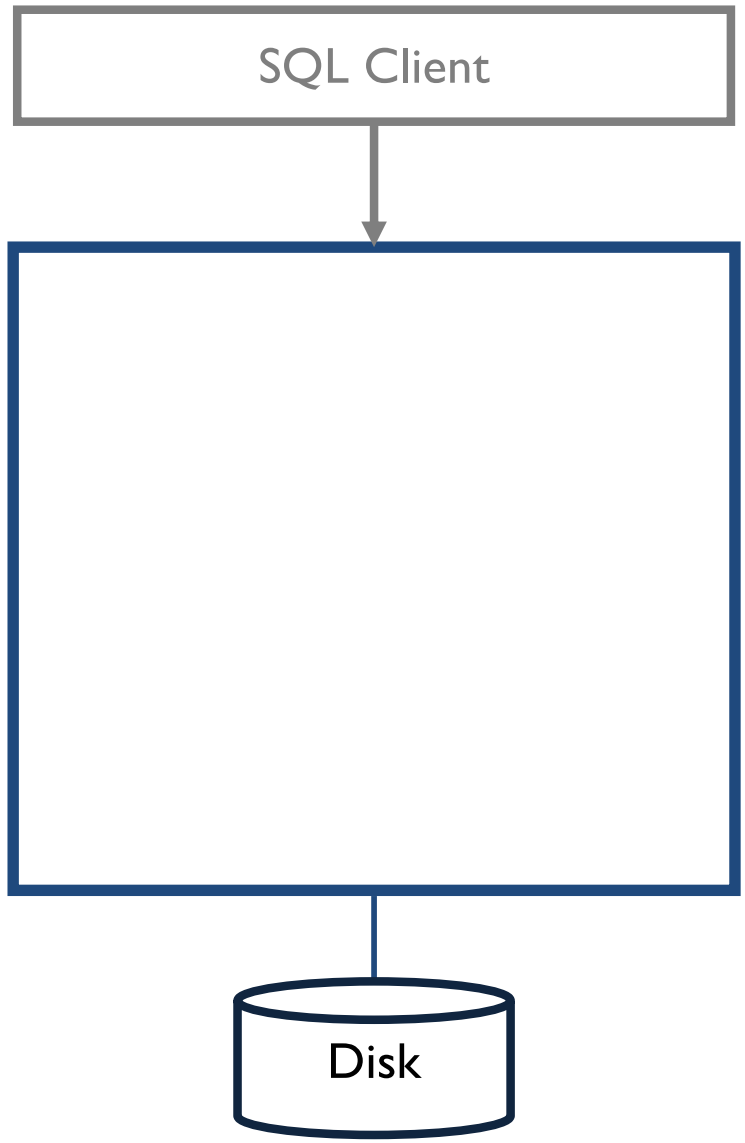
Requires careful design and assumptions for performance



DBMS Overview

Apps interact w/ SQL Client

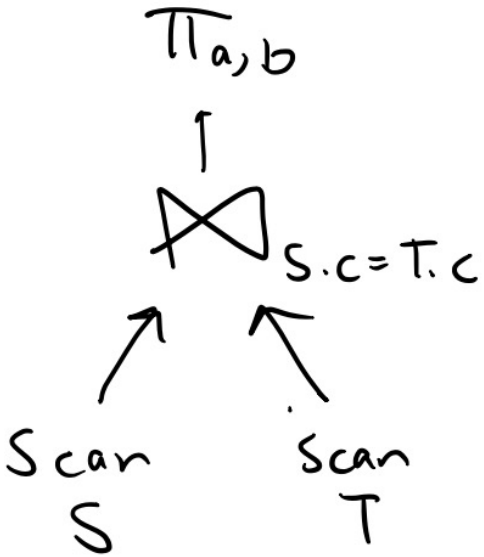
```
db.execute(‘‘‘
  SELECT a, b
    FROM S, T
   WHERE S.c = T.c
’’’)
```

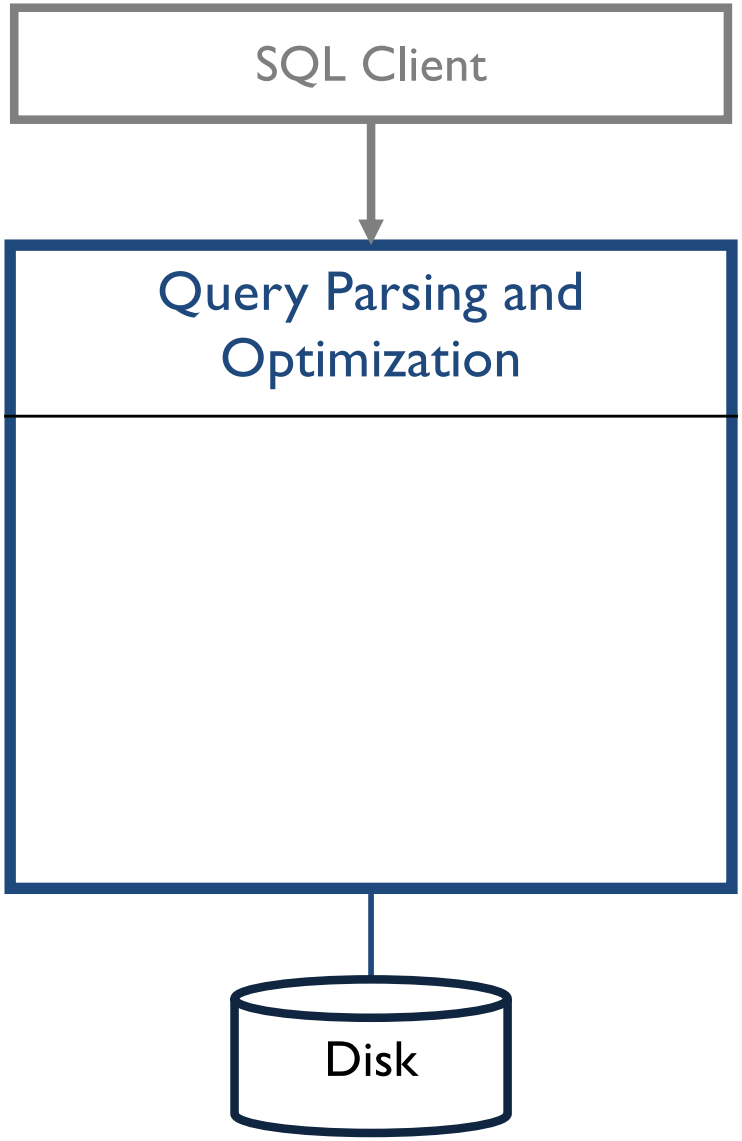


DBMS Overview

Parse, check, & verify SQL query

Turn into efficient query plan composed of logical operators

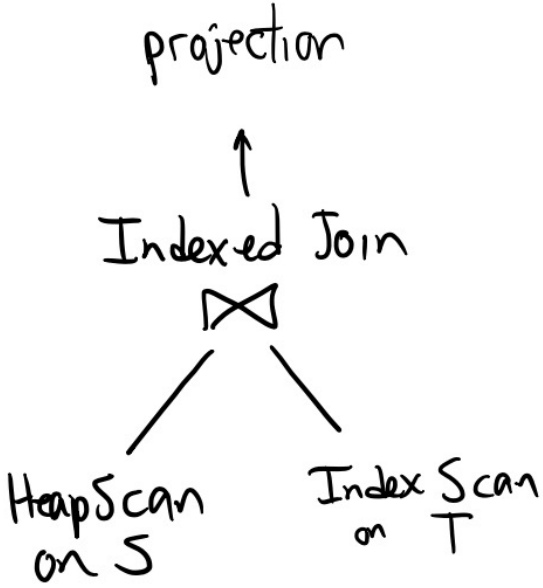


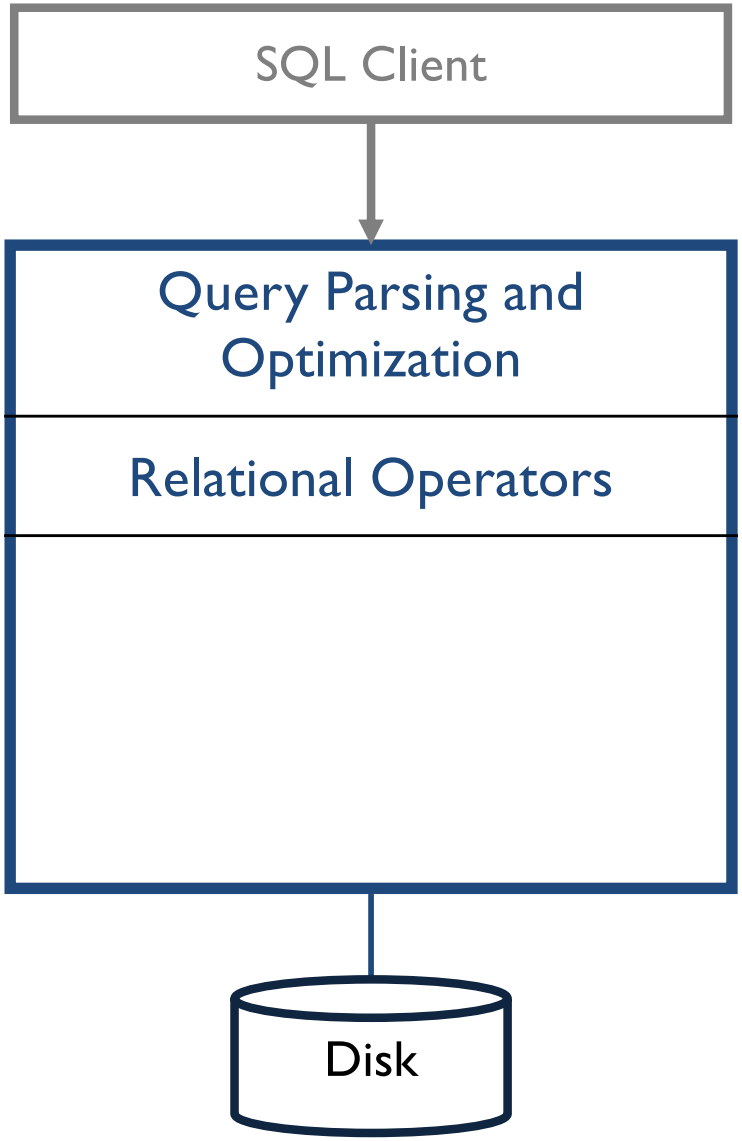


DBMS Overview

Query is modeled as a relational data flow

Each operator is a specific implementation





DBMS Overview

Organizes tables, indexes, records as groups of pages in a "logical file"

API:

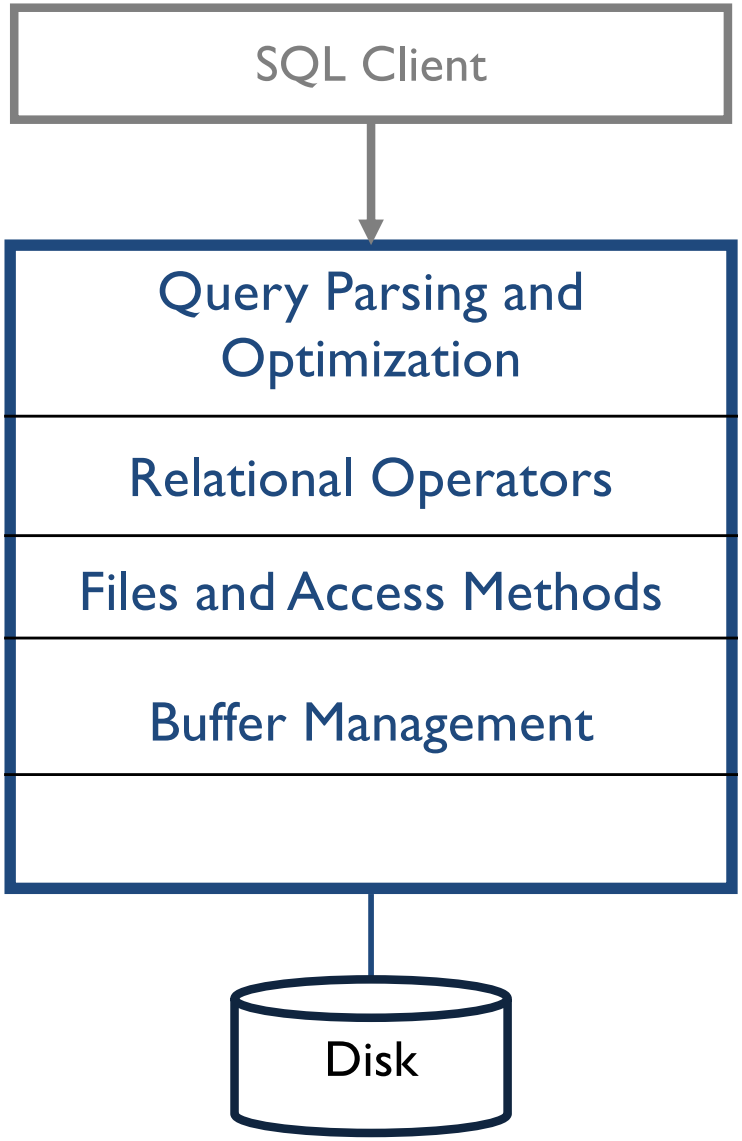
- Operators ask for records
- Logical files help read and write bytes on pages

DBMS Overview

Not all pages can fit into RAM.

Buffer manager provides illusion that all pages are accessible.

Files simply ask for pages.

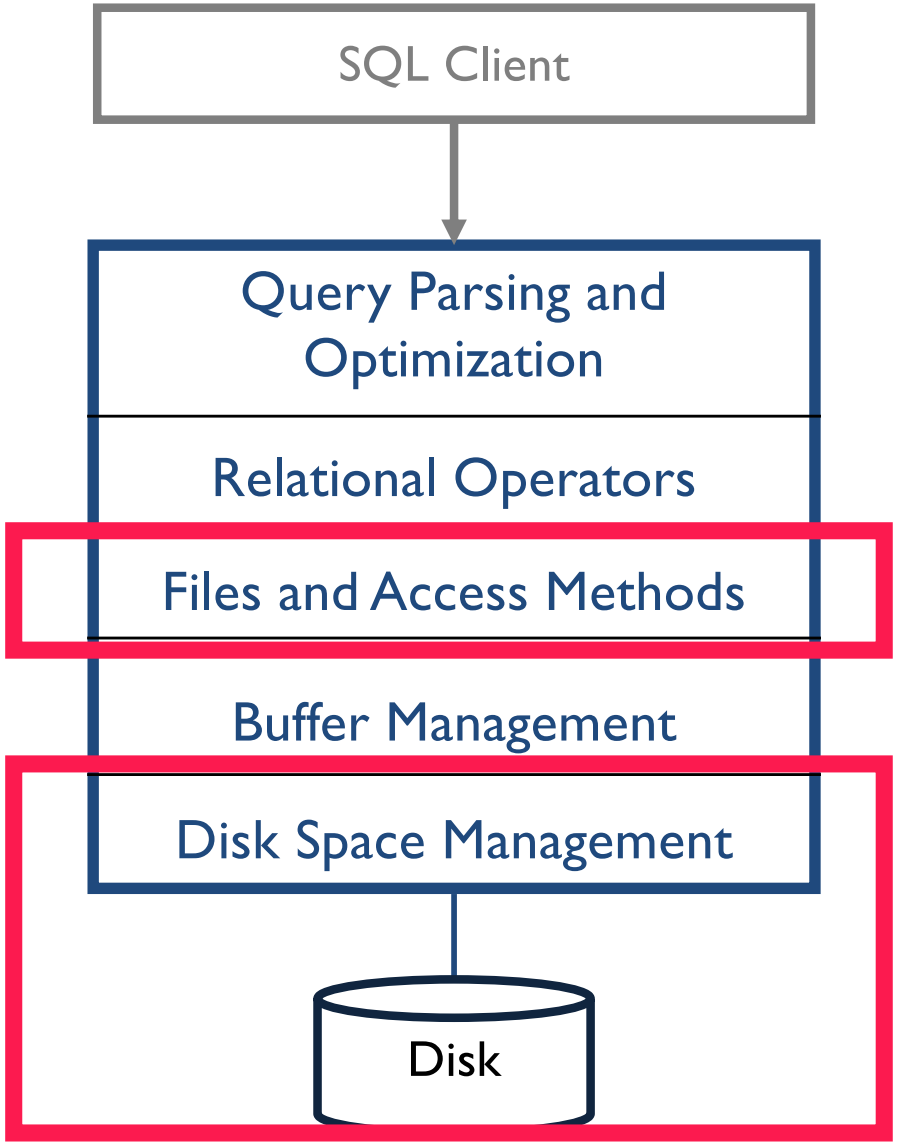


DBMS Overview

Physically read and write bytes on one or more storage devices

(hard drives, SSDs, etc)

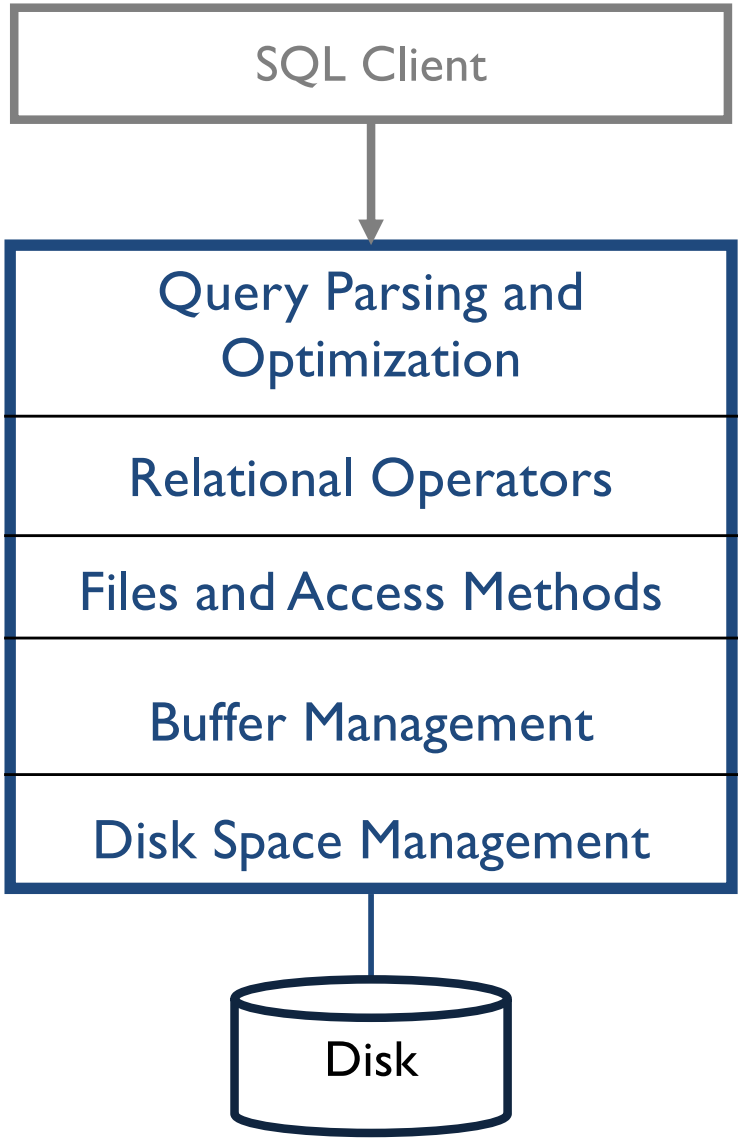
Storage performance properties dictate the design of layers above.



DBMS Overview

Layers help manage engineering complexity.

Requires assumptions about performance of lower layers. (cost models)



Storage Devices

Hard Drives vs SSDs vs RAM

Locality (random vs sequential)

Reads vs Writes

Performance and Monetary Costs

Why not store all in RAM? $$$

Too much $$$

- High-end DBs: ~Petabyte (1000TB).
- SQL Hyperscale: 100TB+ (2018)
- Disks are ~60% cost of a production system

Main memory not persistent

- Obviously important if DB stops/crashes

main-memory DBMSes discussed in advanced DB course

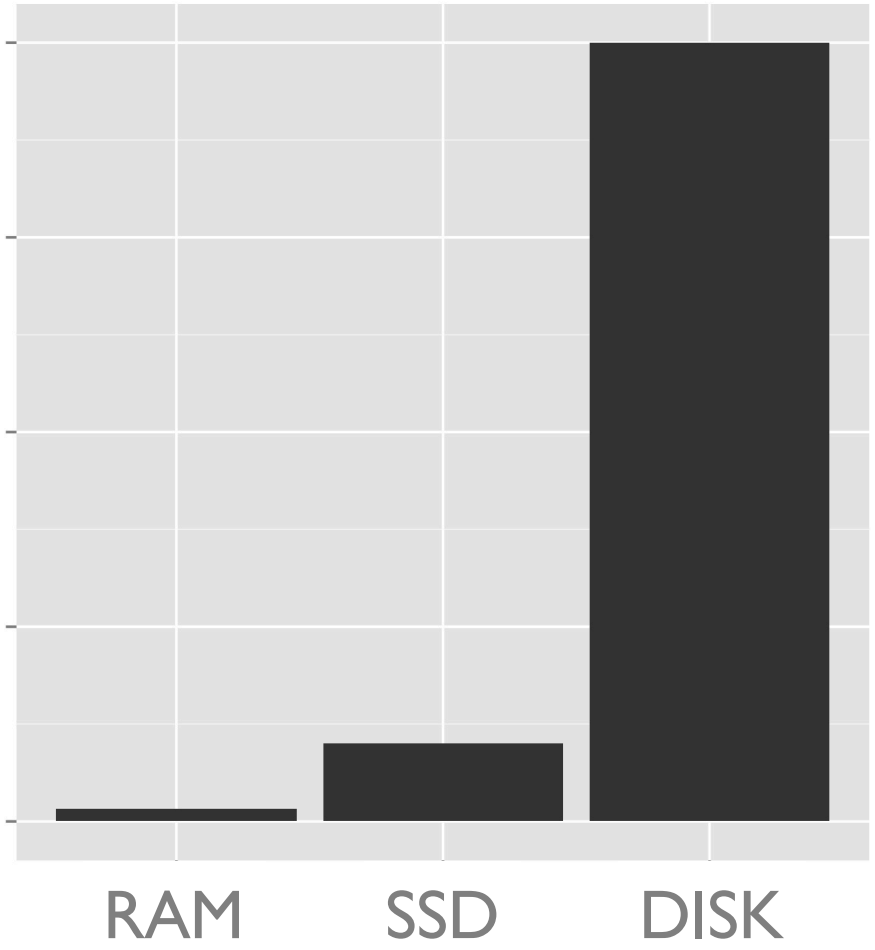
$ Matters

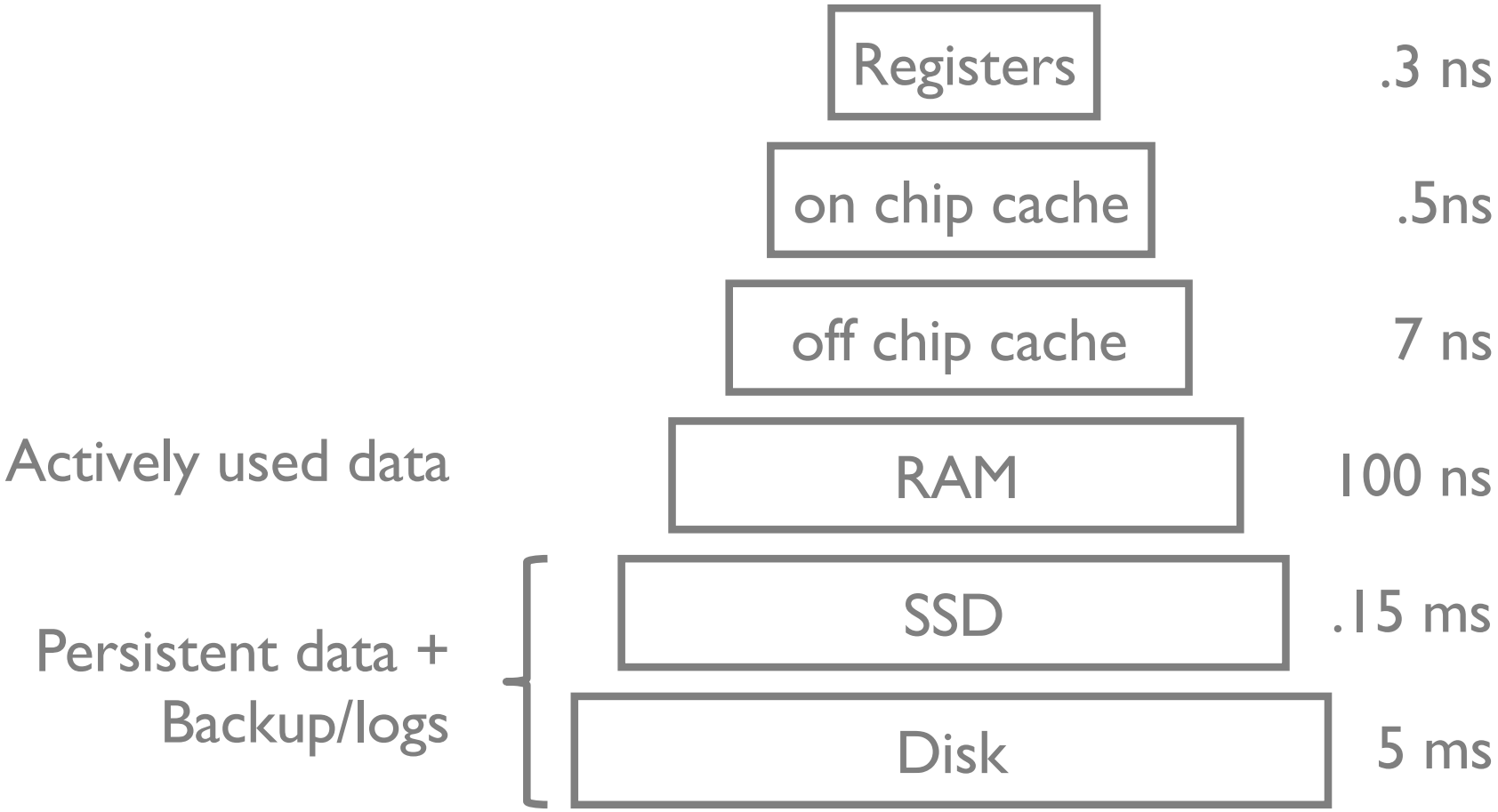
Newegg enterprise $1000

RAM: 0.08TB

SSD: ~2TB (25x)

Disk: ~40TB (500x)

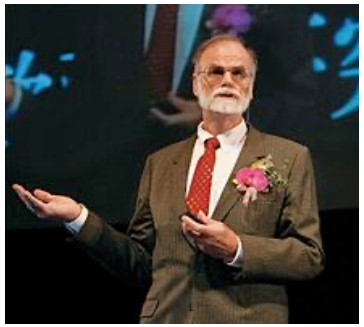


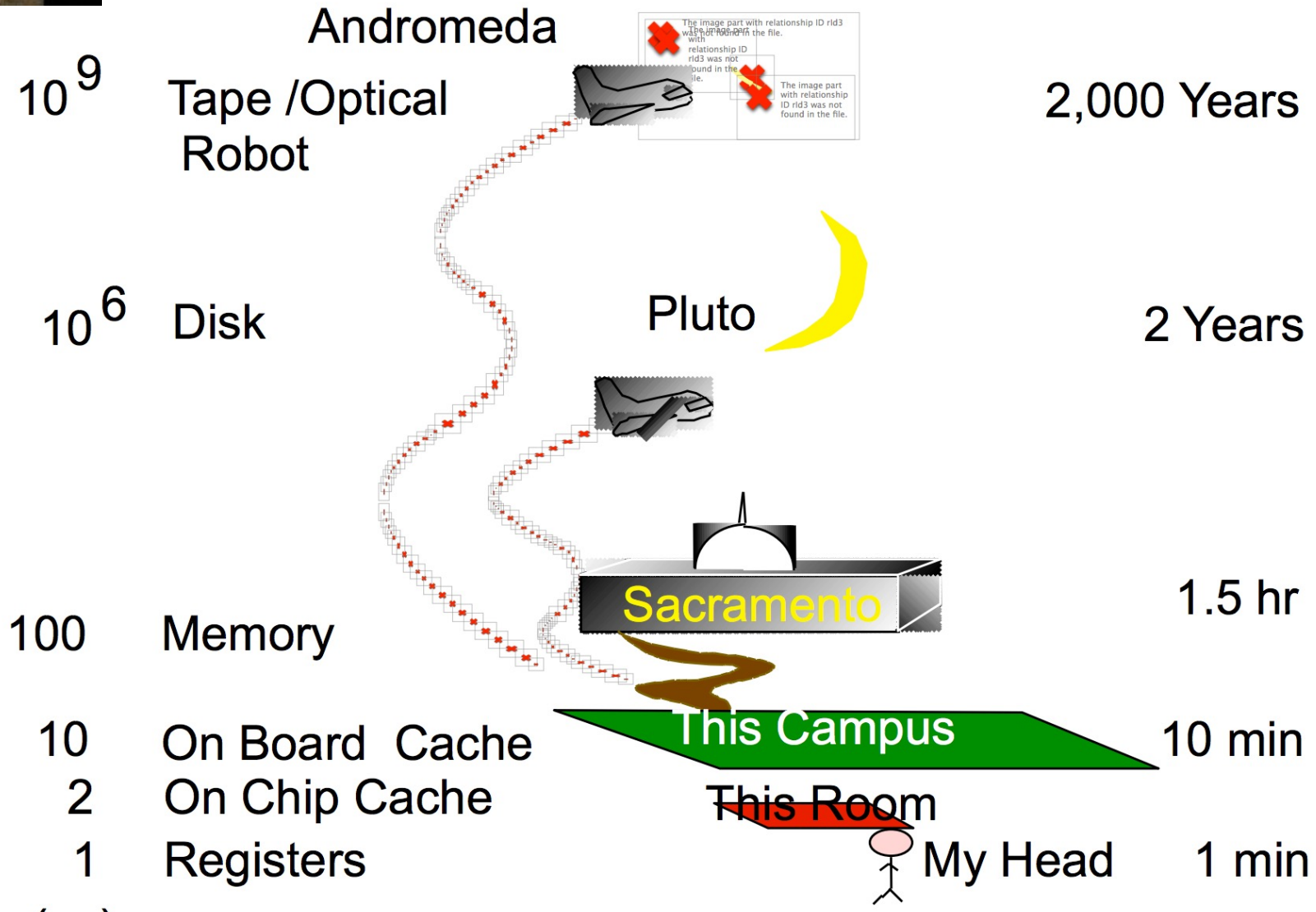


Interesting numbers

compress 1k bytes:	3000 ns
roundtrip in data center:	.5 ms

https://gist.github.com/jboner/2841832



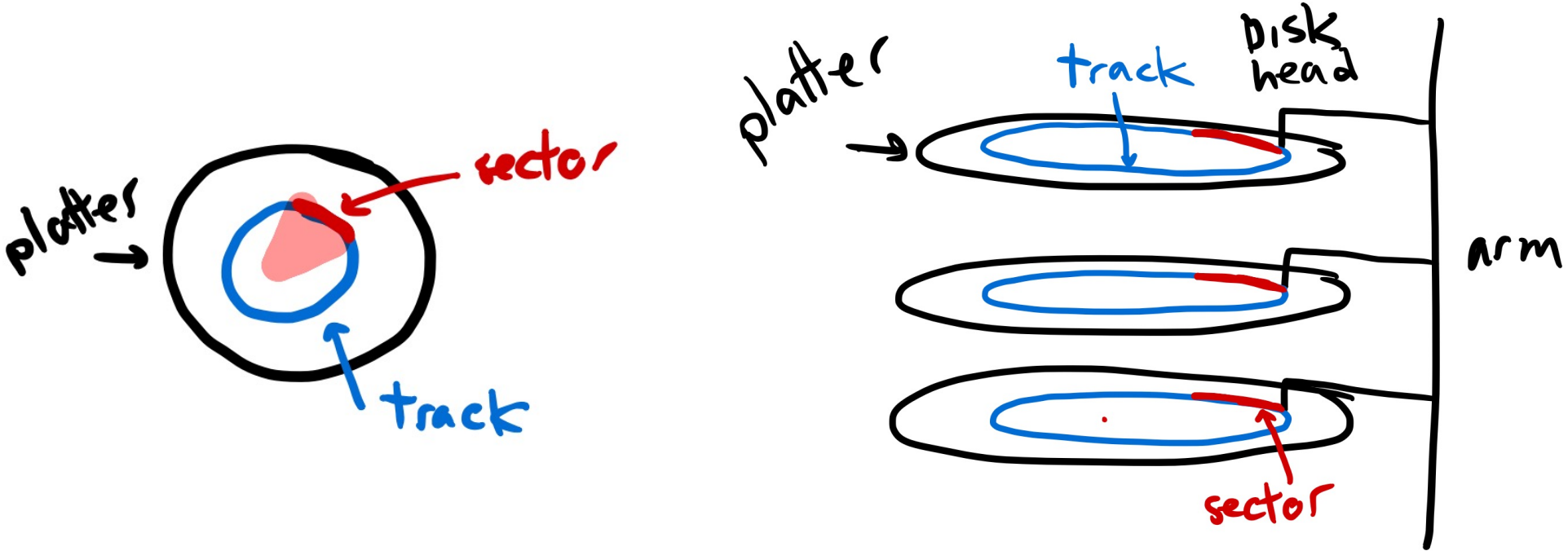

Andromeda
10 9
Tape /Optical
Robot
2,000 Years
10 6
Disk
Pluto
2 Years
Sacramento
1.5 hr
100
Memory
10
On Board Cache
This Campus
10 min
2
On Chip Cache
This Room
1
Registers
My Head
1 min
(ns)

Spin speed: ~7200-15K RPM. 10K RPM ~ 6ms/rotation

Sector: multiples of pages.

More sectors further from center

Terminology: block = page in this class



https://www.youtube.com/watch?v=BIB49F6ExkQ

Time to access (read or write) a disk block

seek time	2-4 msec avg
rotational delay	2-4 msec
transfer time	0.3 msec/64kb page

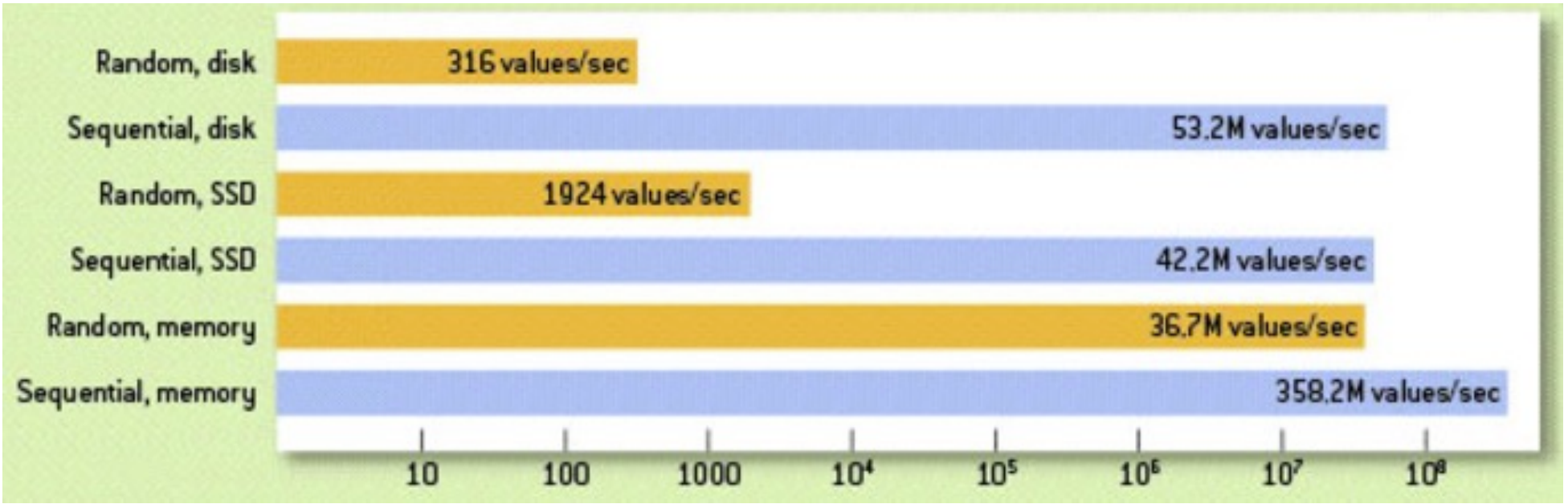
Throughput

read	~150 MB/sec
write	~50 MB/sec

Key: reduce seek and rotational delays

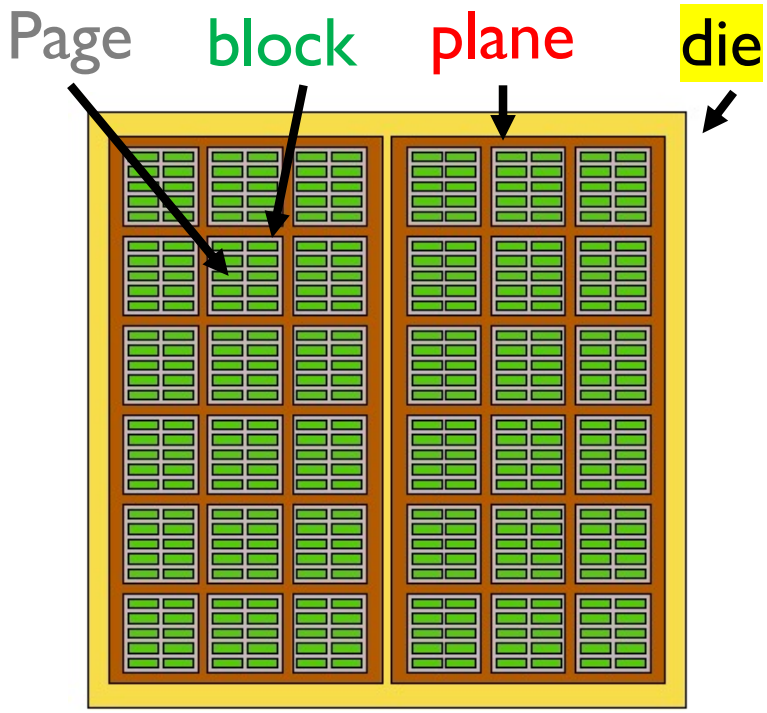
HW & SW approaches

of 4 byte values read per second



5 orders of magnitude

SSDs



NAND memory

Small reads: 4-8k sized page

Big writes: 1-2MB block

2-3k writes before failure

wear leveling: distribute writes around

Write amplification: changing 4 bytes writes a 1-2MB chunk!

Need to think about wear, garbage collection, writing in bulk

https://www.anandtech.com/show/2738/5

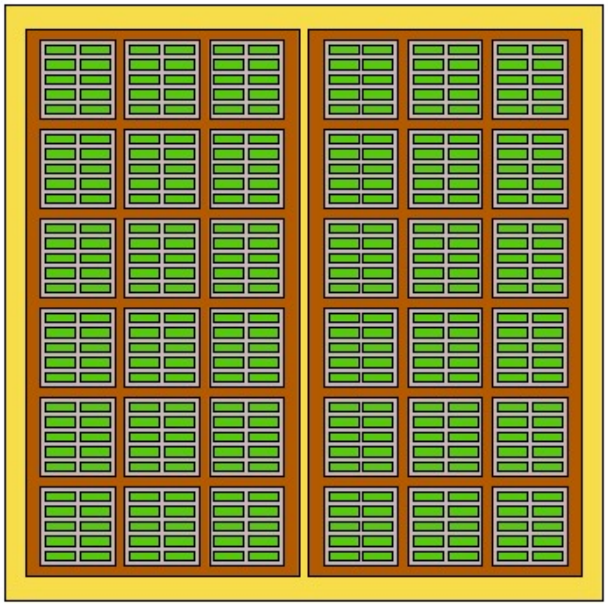
SSDs

Reads fast

- single read time: 0.03ms
- random reads: 500MB/s
- sequential reads: 525MB/s

Writes less predictable

- single write time: 0.03ms
- random writes: 120MB/s
- sequential writes: 480MB/s



What's Best? Depends on Application

Small databases: SSD/RAM

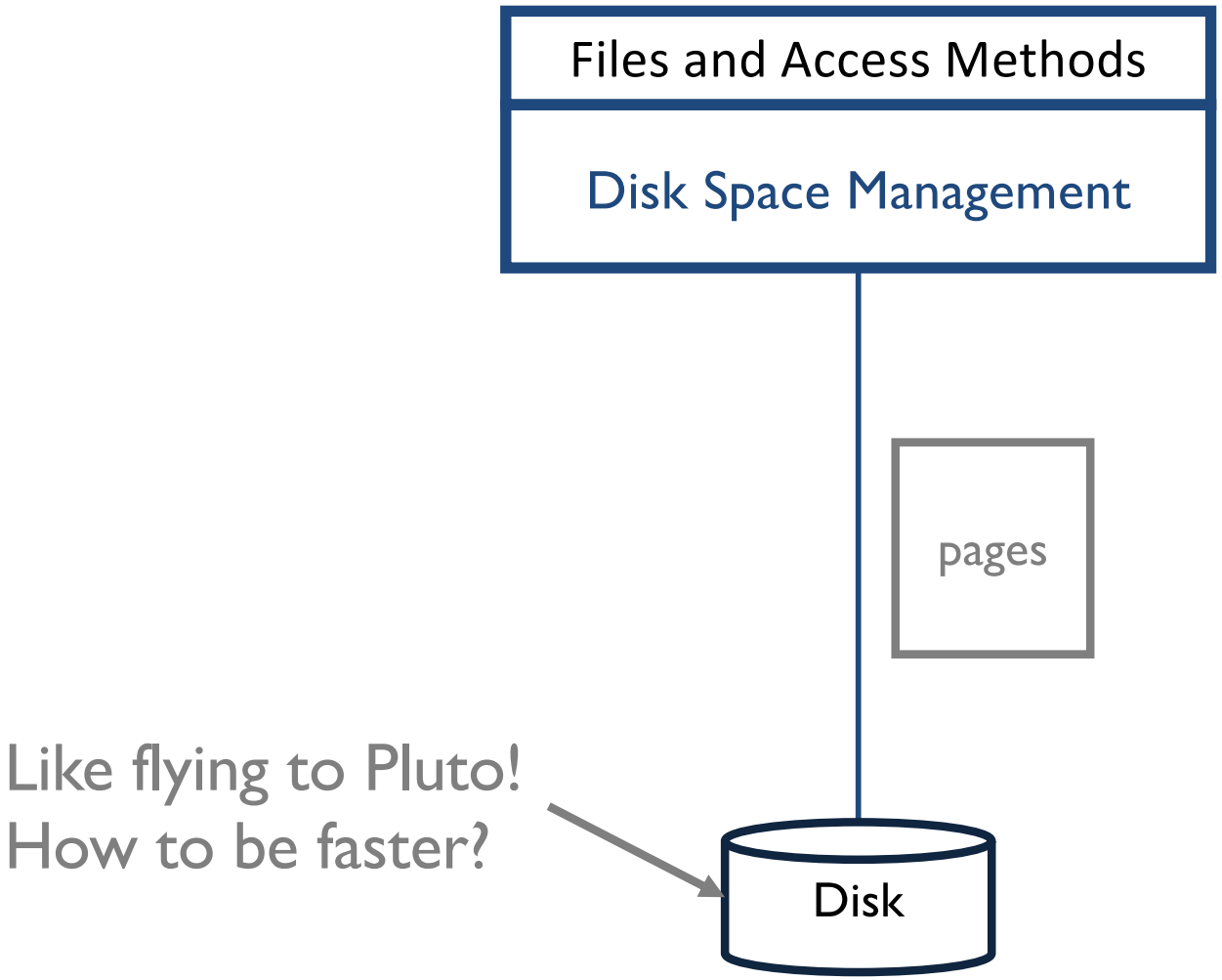
- All global daily weather since 1929: 20GB
- 2000 US Census: 200GB
- 2009 english wikipedia: 14GB
- Easily fits on an SSD or in RAM

Big databases: Disk

- Sensors easily generate TBs of data/day
- Boeing 787 generates ½ TB per flight
- Disk has best cost-capacity ratio
- SSDs help reduce read variance

AWS h1.16xlarge: 64 vCPU, 256GB ram, 8x2T hard drives

Work from the bottom up



Strategies for Fast Data Access

(think about going to the store)

Big difference between random & sequential access

- Optimize for sequential accesses

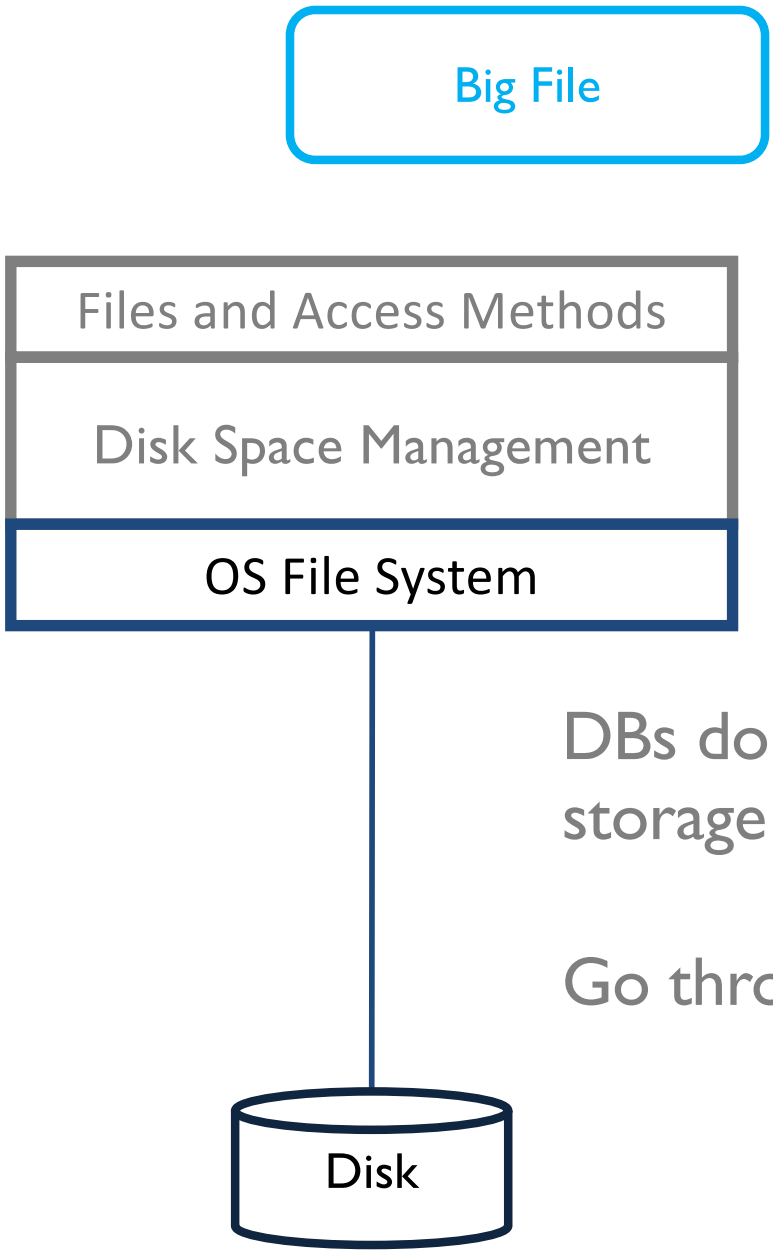
Amortize sequentially read & write big chunks of bytes

Cache popular blocks

Pre-fetch what you will need later

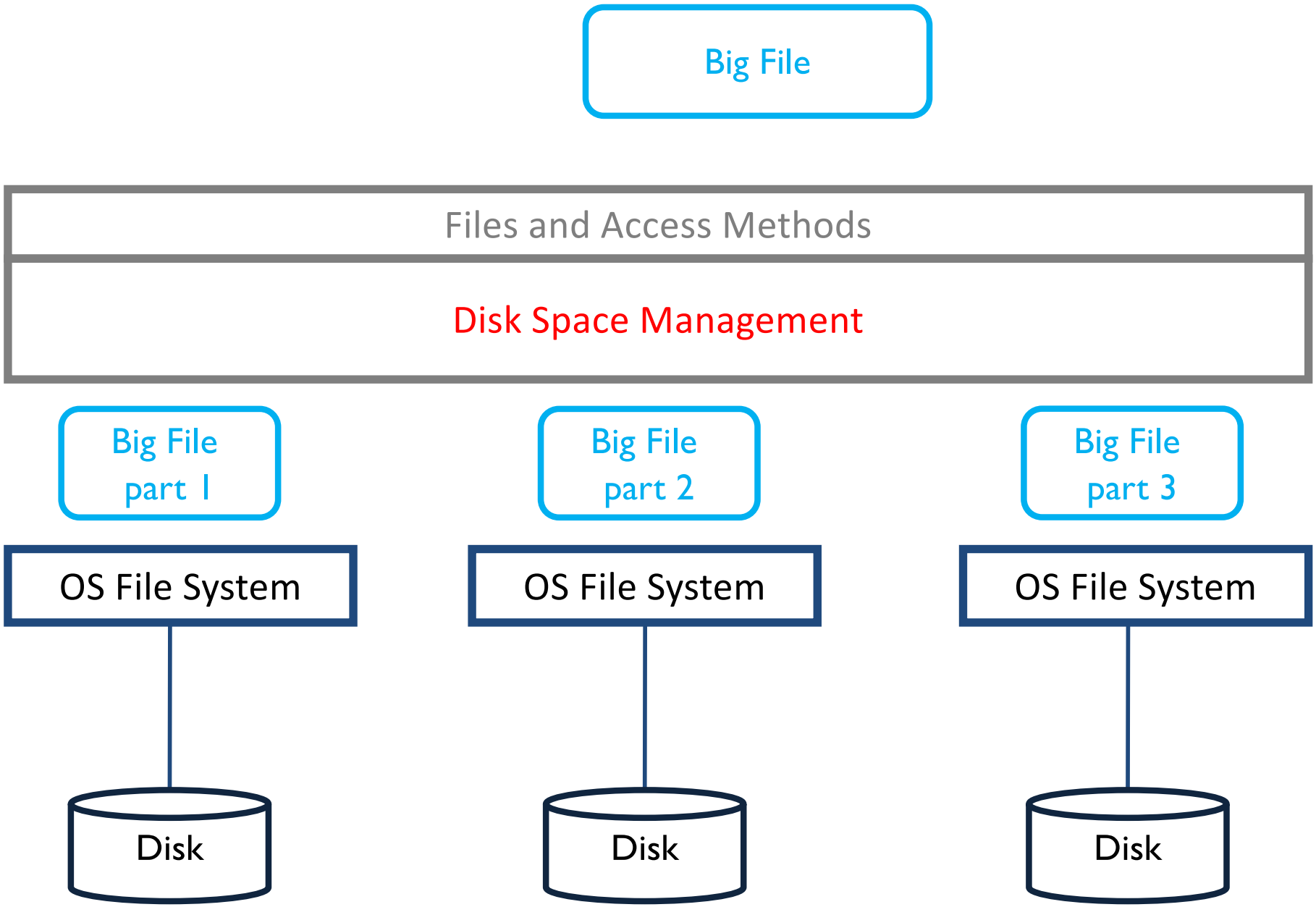
API

- read/write page
- read/write sequential pages
- notion of "next" page (upper layers can assume next is faster)



DBs don't directly talk to storage devices.

Go through OS file system


Big File
Files and Access Methods
Disk Space Management
Big File part 1
Big File part 2
Big File part 3
OS File System
OS File System
OS File System
Disk
Disk
Disk

PostgreSQL database stored in files

```
ewu@dyn-160-39-248-197 ~/d/p/base> cd 33883/
ewu@dyn-160-39-248-197 ~/d/p/b/33883> ls
112             12526           2601            2612            2659            2696            2840_vm         3456_fsm        3602
113             12528           2601_fsm        2612_fsm        2660            2699            2841            3456_vm         3602_fsm
1247            12529           2601_vm         2612_vm         2661            2701            2995            3466            3602_vm
1247_fsm        12529_fsm       2602            2613            2662            2702            2995_vm         3466_vm         3603
1247_vm         12529_vm        2602_fsm        2613_vm         2663            2703            2996            3467            3603_fsm
1249            12531           2602_vm         2615            2664            2704            3079            3468            3603_vm
1249_fsm        12533           2603            2615_fsm        2665            2753            3079_fsm        3501            3604
1249_vm         12534           2603_fsm        2615_vm         2666            2753_fsm        3079_vm         3501_vm         3605
12504           12536           2603_vm         2616            2667            2753_vm         3080            3502            3606
12504_fsm       12538           2604            2616_fsm        2668            2754            3081            3503            3607
12504_vm        1255            2604_vm         2616_vm         2669            2755            3085            3534            3608
12506           1255_fsm        2605            2617            2670            2756            3118            3541            3609
12508           1255_vm         2605_fsm        2617_fsm        2673            2757            3118_vm         3541_fsm        3712
12509           1259            2605_vm         2617_vm         2674            2830            3119            3541_vm         3764
12509_fsm       1259_fsm        2606            2618            2675            2830_vm         3164            3542            3764_fsm
12509_vm        1259_vm         2606_fsm        2618_fsm        2678            2831            3256            3574            3764_vm
12511           1417            2606_vm         2618_vm         2679            2832            3256_vm         3575            3766
12513           1417_vm         2607            2619            2680            2832_vm         3257            3576            3767
12514           1418            2607_fsm        2619_fsm        2681            2833            3258            3576_vm         548
12514_fsm       1418_vm         2607_vm         2619_vm         2682            2834            3394            3596            549
12514_vm        174             2608            2620            2683            2834_vm         3394_fsm        3596_vm         826
12516           175             2608_fsm        2620_vm         2684            2835            3394_vm         3597            826_vm
12518           2187            2608_vm         2650            2685            2836            3395            3598            827
12519           2328            2609            2651            2686            2836_vm         34002           3598_vm         828
12519_fsm       2328_vm         2609_fsm        2652            2687            2837            34004           3599            PG_VERSION
12519_vm        2336            2609_vm         2653            2688            2838            34004_fsm       3600            pg_filenode.map
12521           2336_vm         2610            2654            2689            2838_fsm        34004_vm        3600_fsm        pg_internal.init
12523           2337            2610_fsm        2655            2690            2838_vm         34008           3600_vm
12524           2600            2610_vm         2656            2691            2839            34031           3601
12524_fsm       2600_fsm        2611            2657            2692            2840            3455            3601_fsm
12524_vm        2600_vm         2611_vm         2658            2693            2840_fsm        3456            3601_vm
ewu@dyn-160-39-248-197 ~/d/p/b/33883>
```

Administrivia

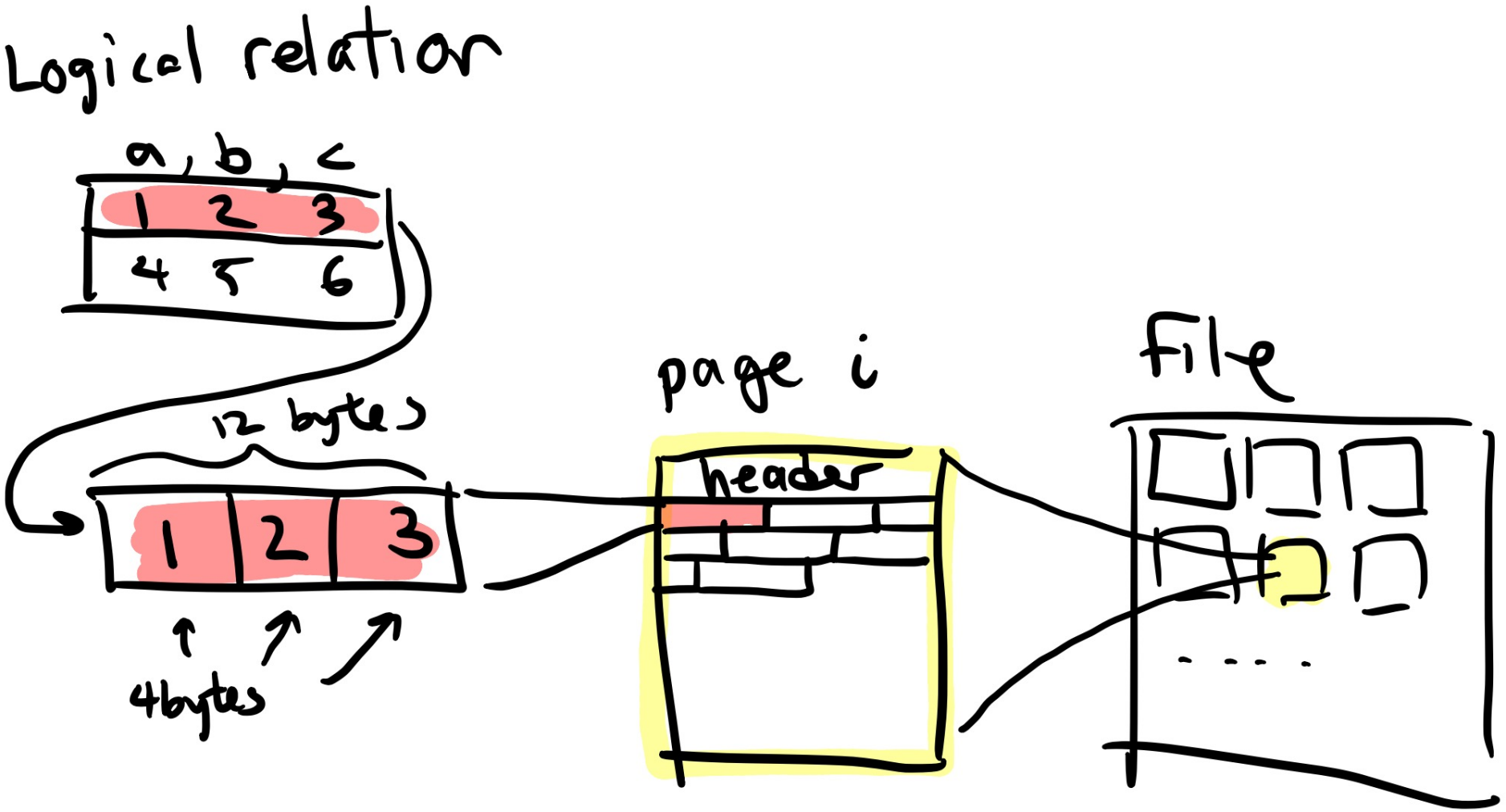
HW4 out

Project 2 out: **Start early!!!!**

Both Due 11/29

Maximum 3 late days!

Files


Logical relation
a, b, c
1 2 3
4 5 6
12 bytes
1 2 3
4bytes
page i
header
File

Files

Higher layers want to talk in terms of records, and files of records

File: collection of pages

Minimum API:

- insert/delete/modify record
- lookup record_id
- scan all records

Page: collection of records

- typically *fixed page sizes* (8 or 64kb in PostgreSQL)

These are logical.

File Organizations Overview

What do pages store, and how to organize pages?

- Unordered heap file
- Heap file
- B+ tree index
- Hash index

Units that we'll care about

Ignore CPU cost
Ignore RAM cost

B # *data* pages on disk for relation
R # records per data page
D avg time to read/write data page to/from disk

Simplifies life when computing costs
OK to not be exactly correct

Unordered Heap Files

Collection of records (no order)

As we add records, pages allocated
As we remove records, pages removed

To support record level ops, need to track:
- pages in file
- free space on pages
- records on page

Ok, let's design that

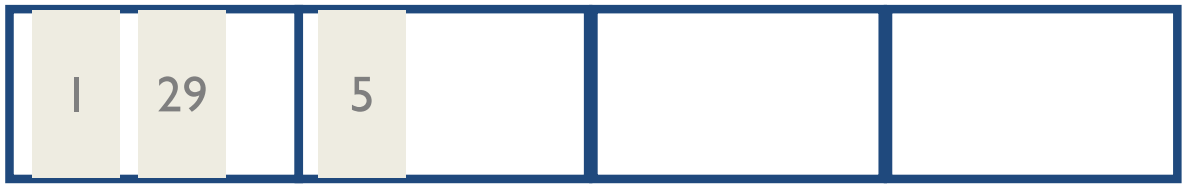
Super Naïve Design

Big array
of bytes

Heap is a big array of bytes.

First split into array of pages

Super Naïve Design



Array of pages

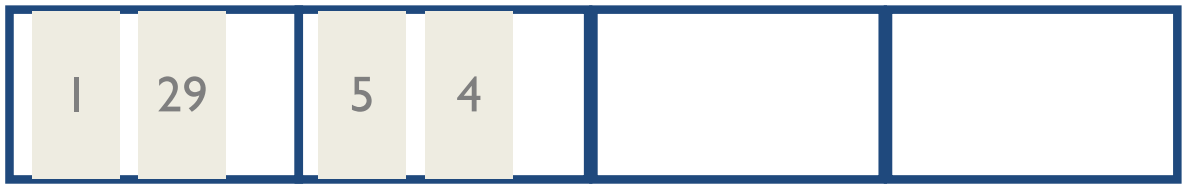
Insert 4

Each each page from disk sequentially

Check if there's an open spot

Insert 4 into open spot.

Super Naïve Design

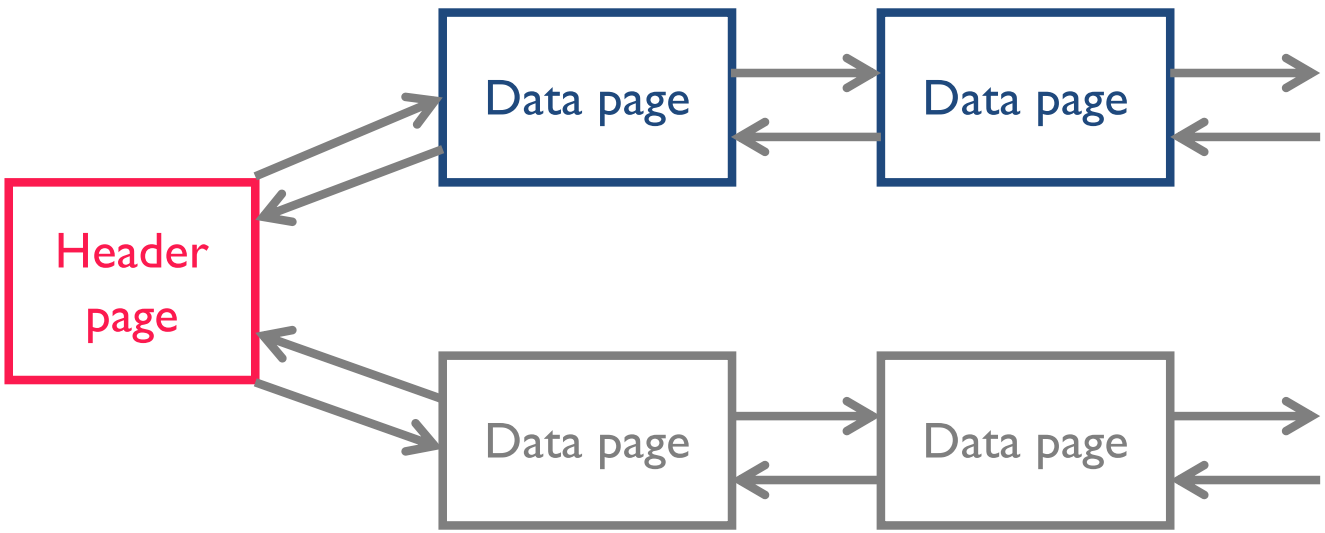


Array of pages

Some problems include

- Slow to know what pages are empty, full, have space
- Slow to find a particular value
- Fragmentation

Heap File



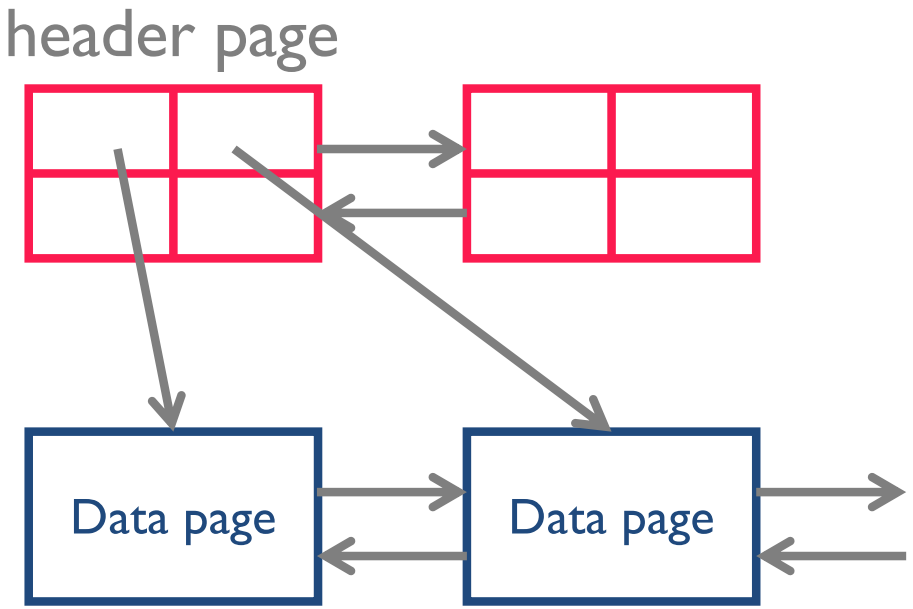
Full data pages

Pages with free space

Header page info stored in catalog
Data page contains: 2 pointers, free space, data

Need to scan pages to answer any query
Which page has enough free space for 100 bytes?

Use a directory

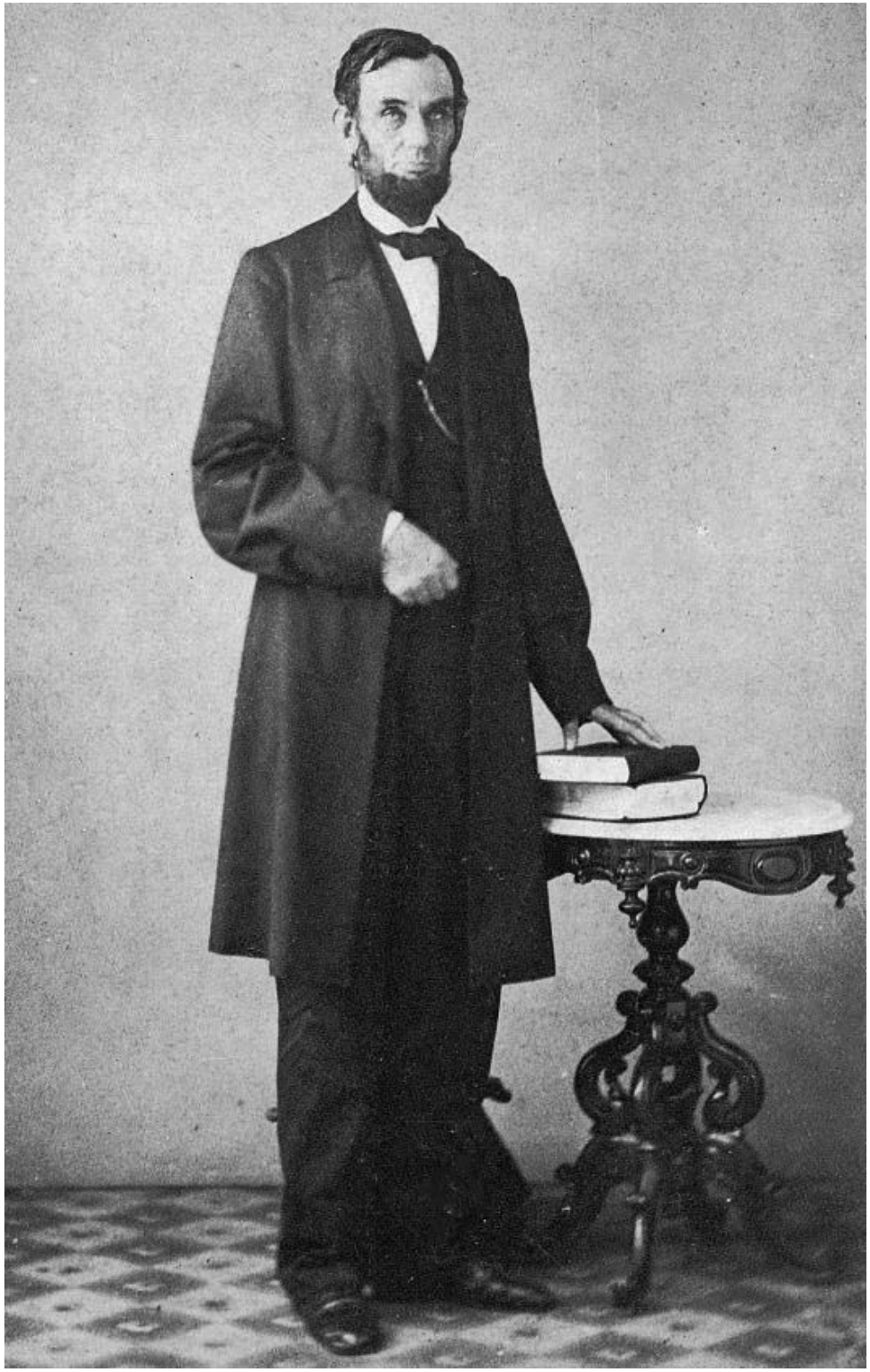


Directory

All data pages

Directory entries track # free bytes on data pages

Directory is collection of pages



Indexes

"If I had 8 hours to chop down a tree,

I'd spend 6 sharpening my ax."

Abraham Lincoln

Indexes

Heap files answer any query via a sequential scan, but…

Queries use *qualifications* (predicates)

- find students where class = "CS"
- find students with age > 10

Indexes: file structures for value-based queries

- B+-tree index (~1970s)
- Hash index

Overview! Details in 4112

Indexes

Defined with respect to a *search key*

don't confuse with candidate keys!!

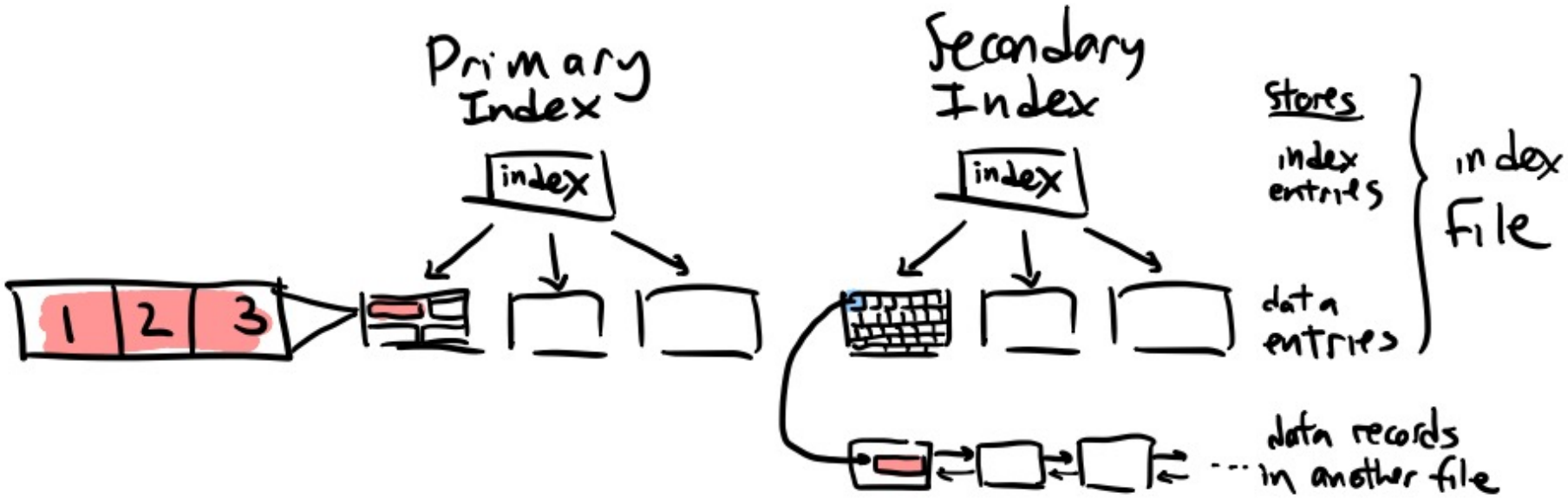
Faster access for WHERE clauses w/ search key

```
CREATE INDEX idx1 ON users USING btree (sid)
CREATE INDEX idx2 ON users USING hash  (sid)
CREATE INDEX idx3 ON users USING btree (age,name)
```

You will play around with indexes in HW4

https://www.postgresql.org/docs/11/sql-createindex.html

Primary vs Secondary Index Files



Primary

Data entries contain actual tuples

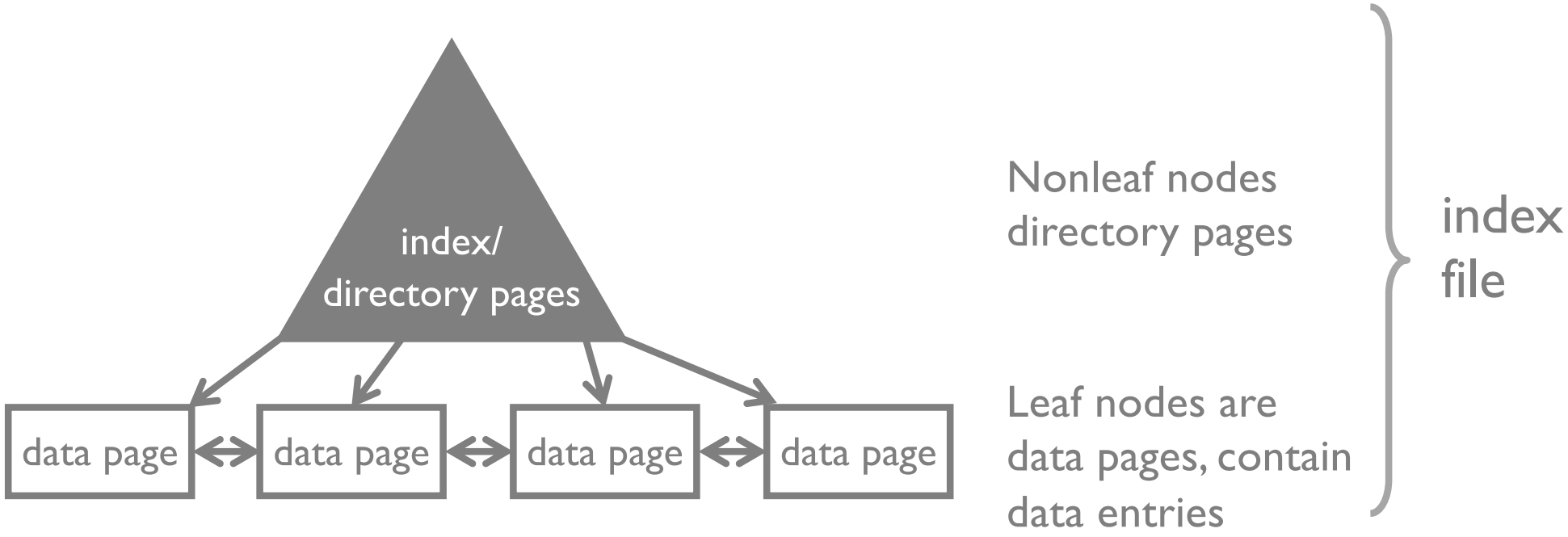
Pros: directly access data.

Secondary

Data entries contain <search key val, rid> pointers into another file

Pros: index is more compact

B+ Tree Index: disk-optimized index



Node = Page

Supports equality and range queries on search key

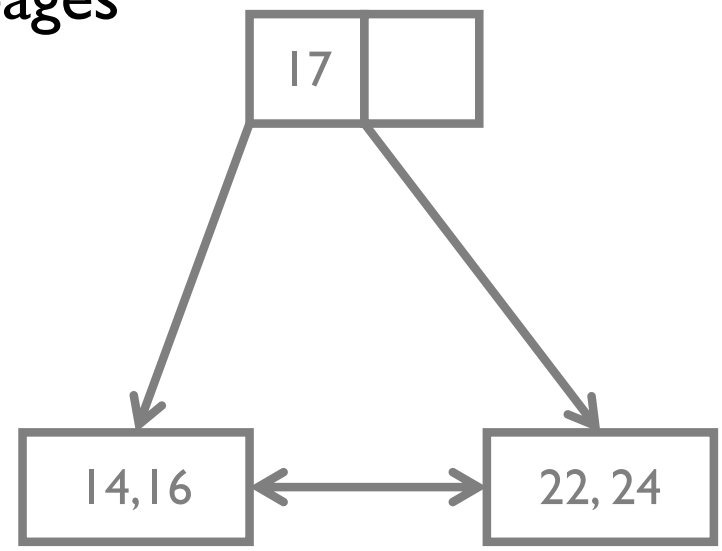
Self balancing (path to any leaf is almost the same)

Leaf nodes are connected via doublely linked list

Height is with respect to directory pages (the gray part of the triangle)

B+ Tree on (age)

Non-leaf Directory pages
m index entries
m+1 pointers

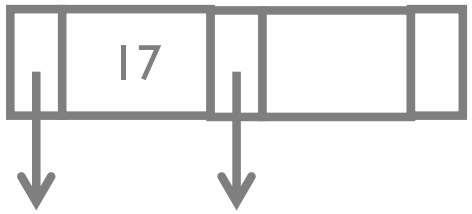


At each level in the tree, index & data page contents are sorted by search key

Leaf Data pages
data entries/tuples

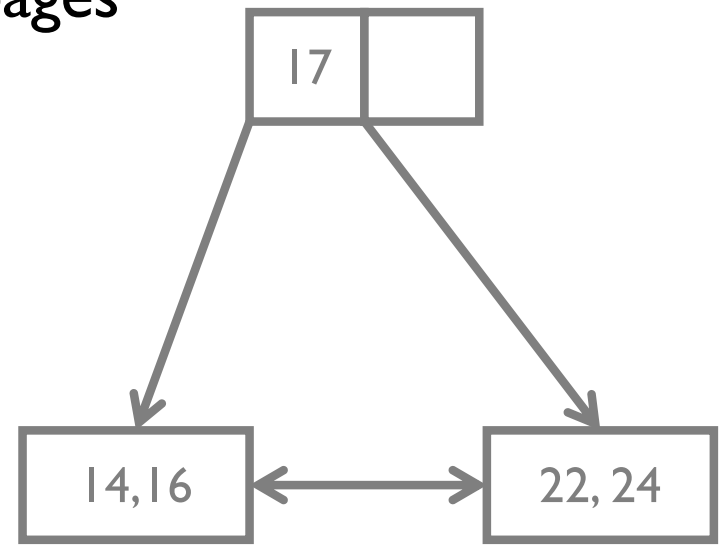
Query: SELECT * WHERE age= 14

directory page



Index Only Queries: B+ Tree on (age)

Non-leaf Directory pages
m index entries
m+1 pointers

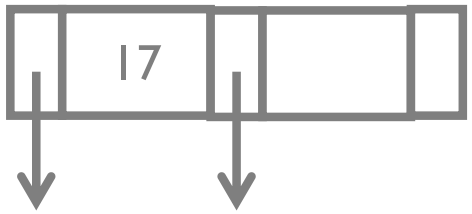


At each level in the tree, index & data page contents are sorted by search key

Leaf Data pages
data entries/tuples

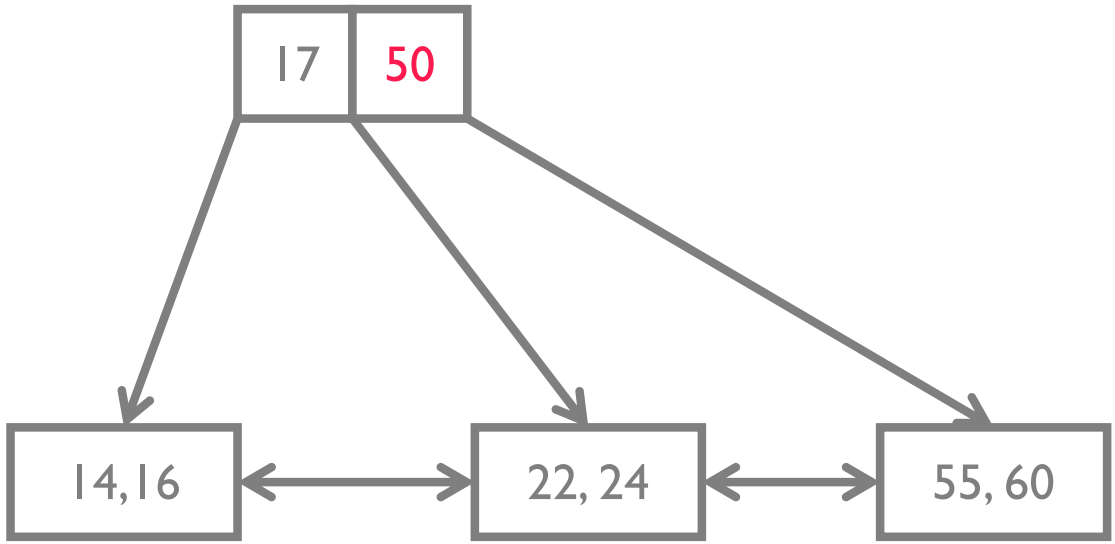
Query: SELECT age WHERE age > 14
(index only!)

directory page



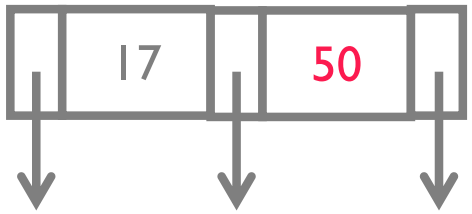
B+ Tree on (age)

Note: 50 not a data entry



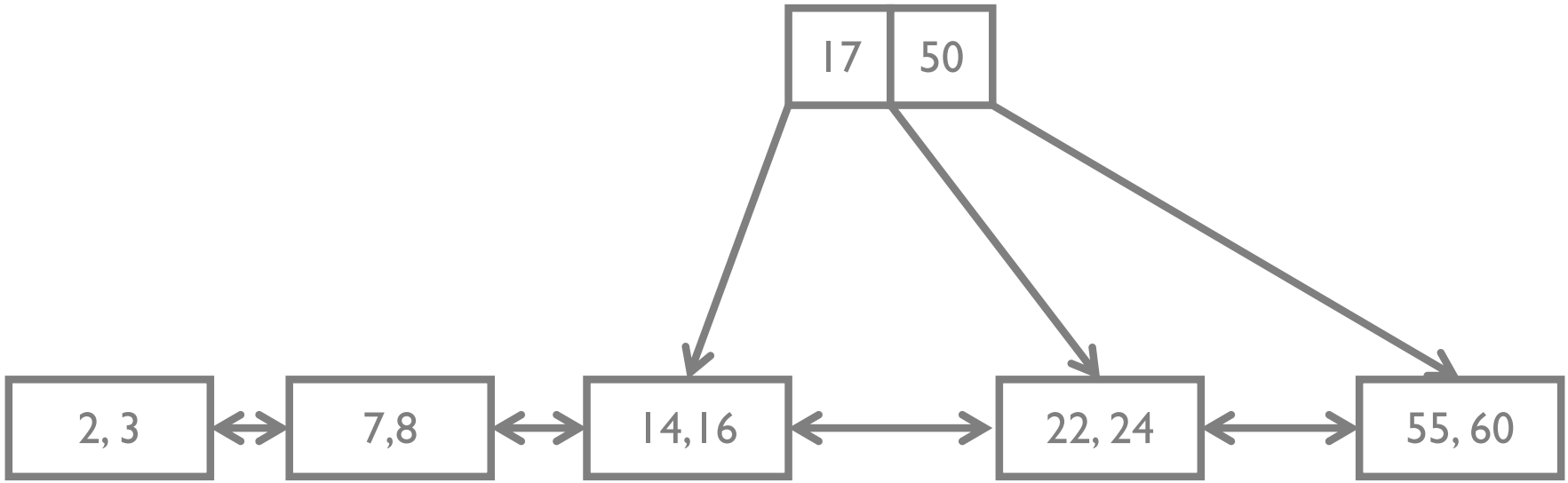
Query: SELECT * WHERE age = 55

directory page



In this class for cost estimation, we will assume number of pointers same as number of directory entries

B+ Tree on (age)

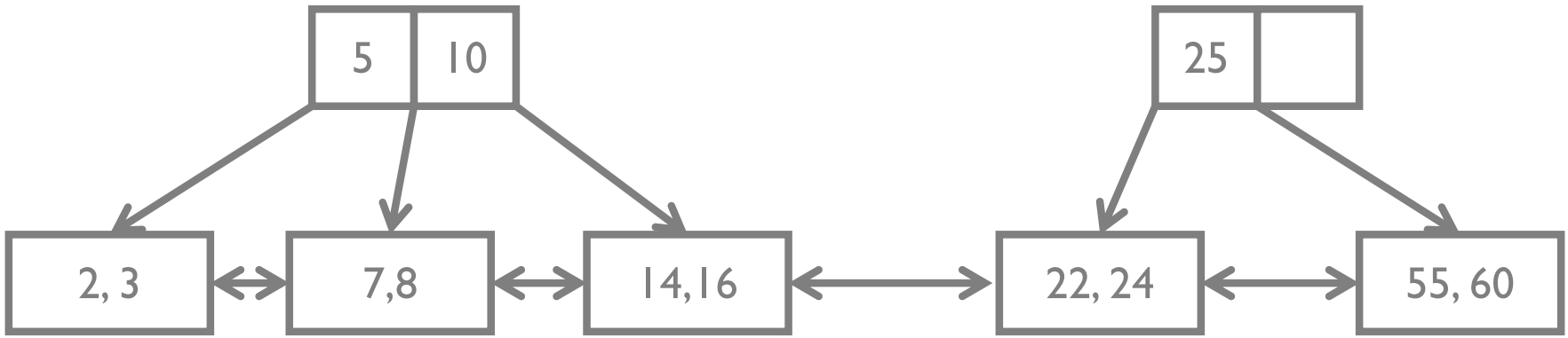


When directory page is full, can't add more pointers

Split directory page and reorganize tree

(don't need to know details of how splits are done)

B+ Tree on (age)

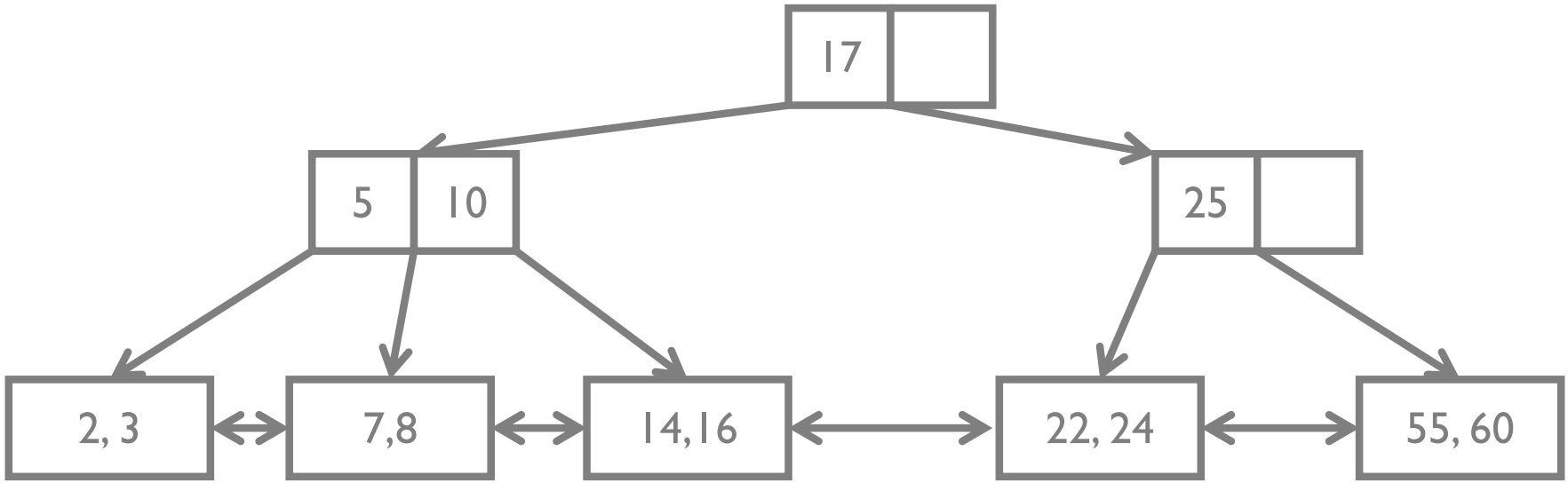


When directory page is full, can't add more pointers

Split directory page and reorganize tree

(don't need to know details of how splits are done)

B+ Tree on (age)

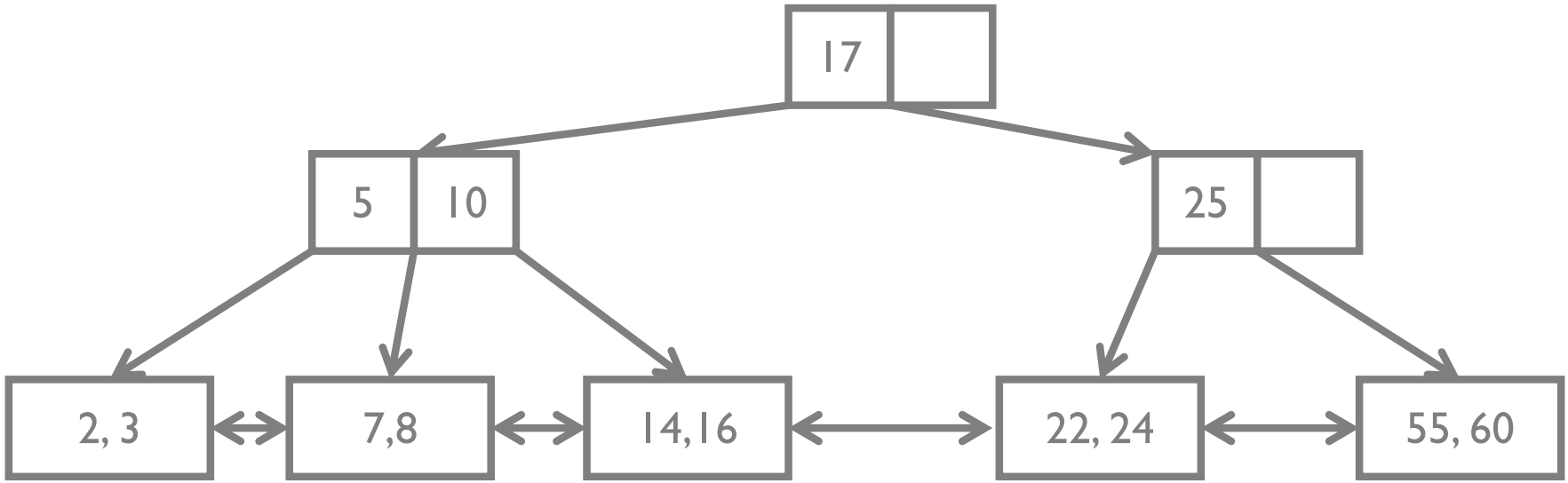


When directory page is full, can't add more pointers

Split directory page and reorganize tree

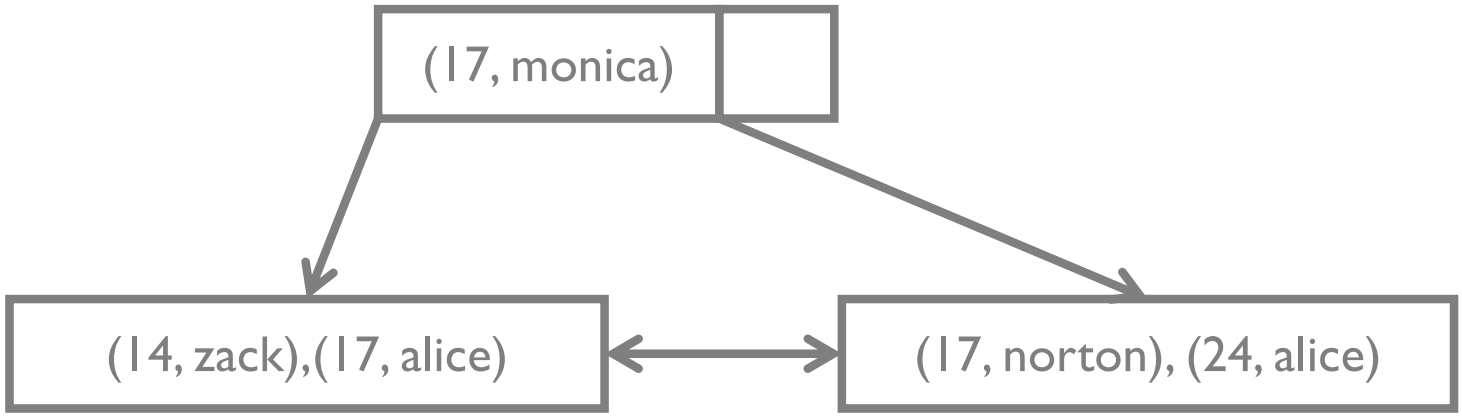
(don't need to know details of how splits are done)

B+ Tree on (age)



Query: SELECT * WHERE age > 20

Composite Keys: B+ Tree on (age, name)

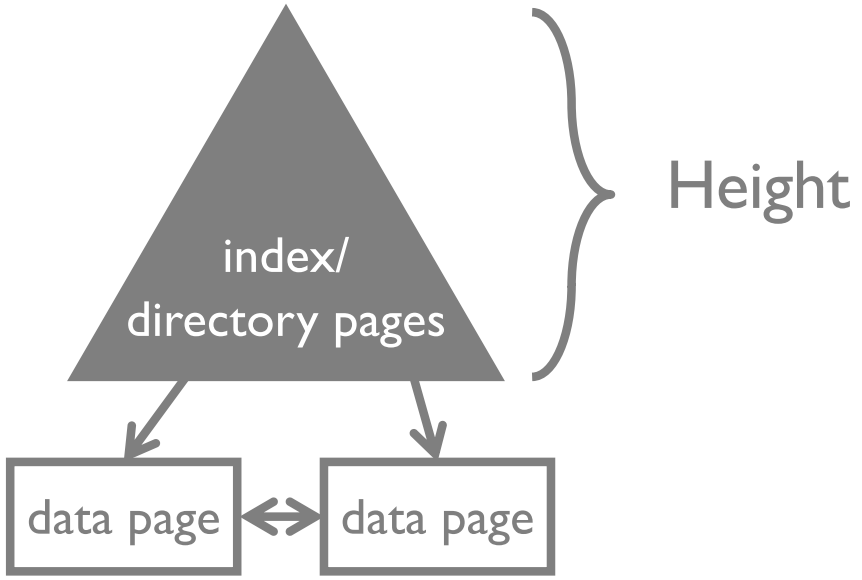


How do the following queries use the index on (age, name)?

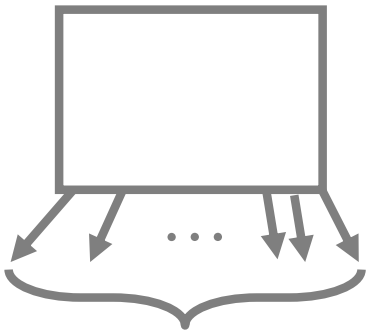
```
SELECT age   WHERE age = 14
SELECT *     WHERE age < 18 AND name < ‘monica’
SELECT age   WHERE name = ‘bobby’
```

Terminology

Keep free space for fast inserts
Reduce "fill factor" parameter



Directory Page



Fanout ("branching factor")

We assume same as # directory entries

Example using 8kb pages

How many entries in the index's data pages?

- fill-factor: ~66%
- ~300 entries per directory page

height 2: 300^3	~	27 Million entries	assuming 300 tuples/pg
height 3: 300^4	~	8.1 Billion entries	assuming 300 tuples/pg

Top levels often in memory

- height 2 only 300 pages ~2.4MB
- height 3 only 90k pages ~750MB

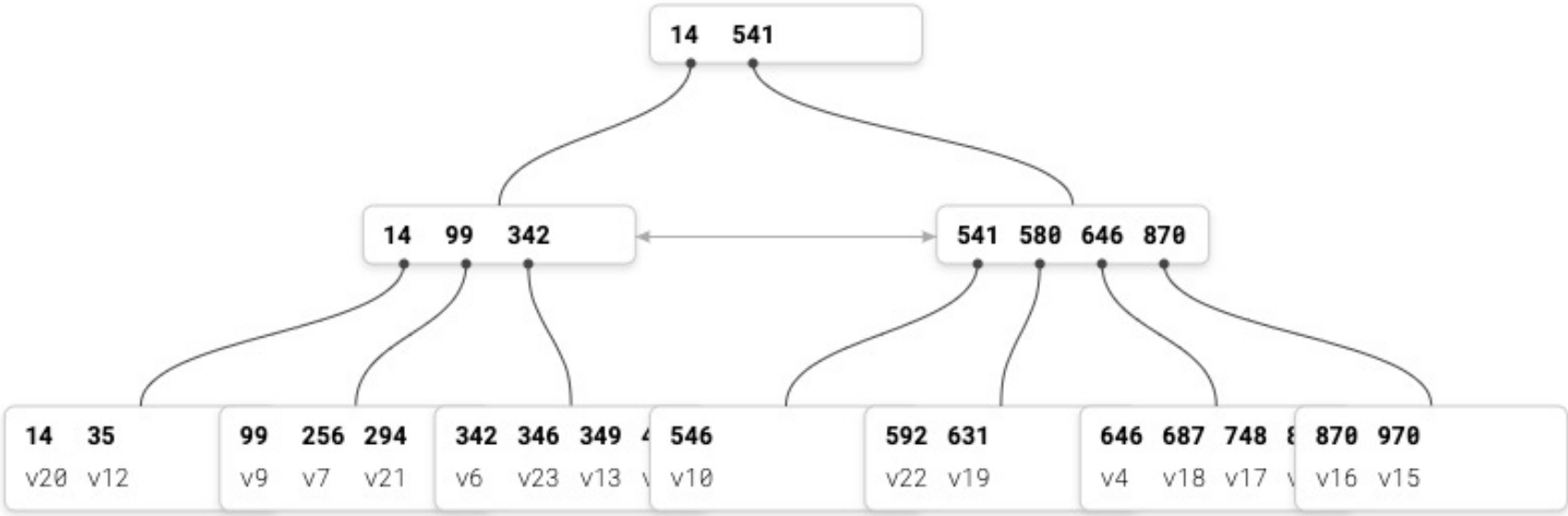
Cool B+Tree viz: https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html

For Real?

problem. We have run TB range databases on Raspberry PI with 4GB of RAM and hammered that in benchmarks (far exceeding the memory capacity). The interesting thing here is that the B+Tree nature means that the upper tiers of the tree were already in memory, so we mostly ended up with a single page fault per request.

https://ayende.com/blog/196161-C/re-are-you-sure-you-want-to-use-mmap-in-your-database-management-system

https://bplustree.app/

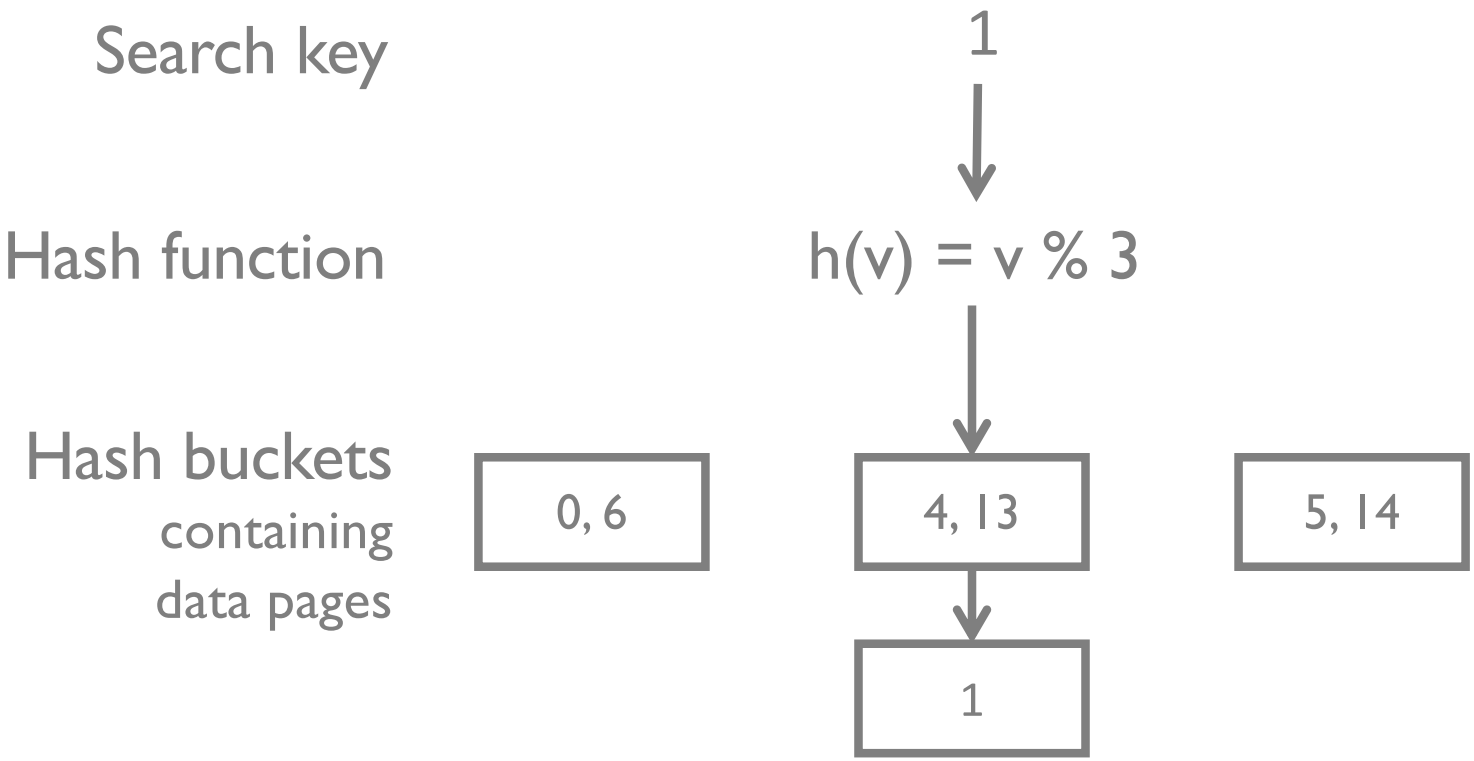


Hash Index on age

Hash function h(v) = v % 3

Hash buckets containing data pages

INSERT Hash Index on age

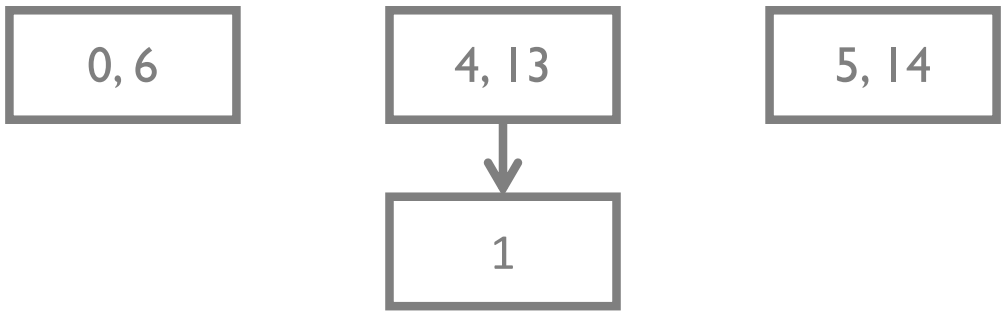


INSERT Hash Index on age

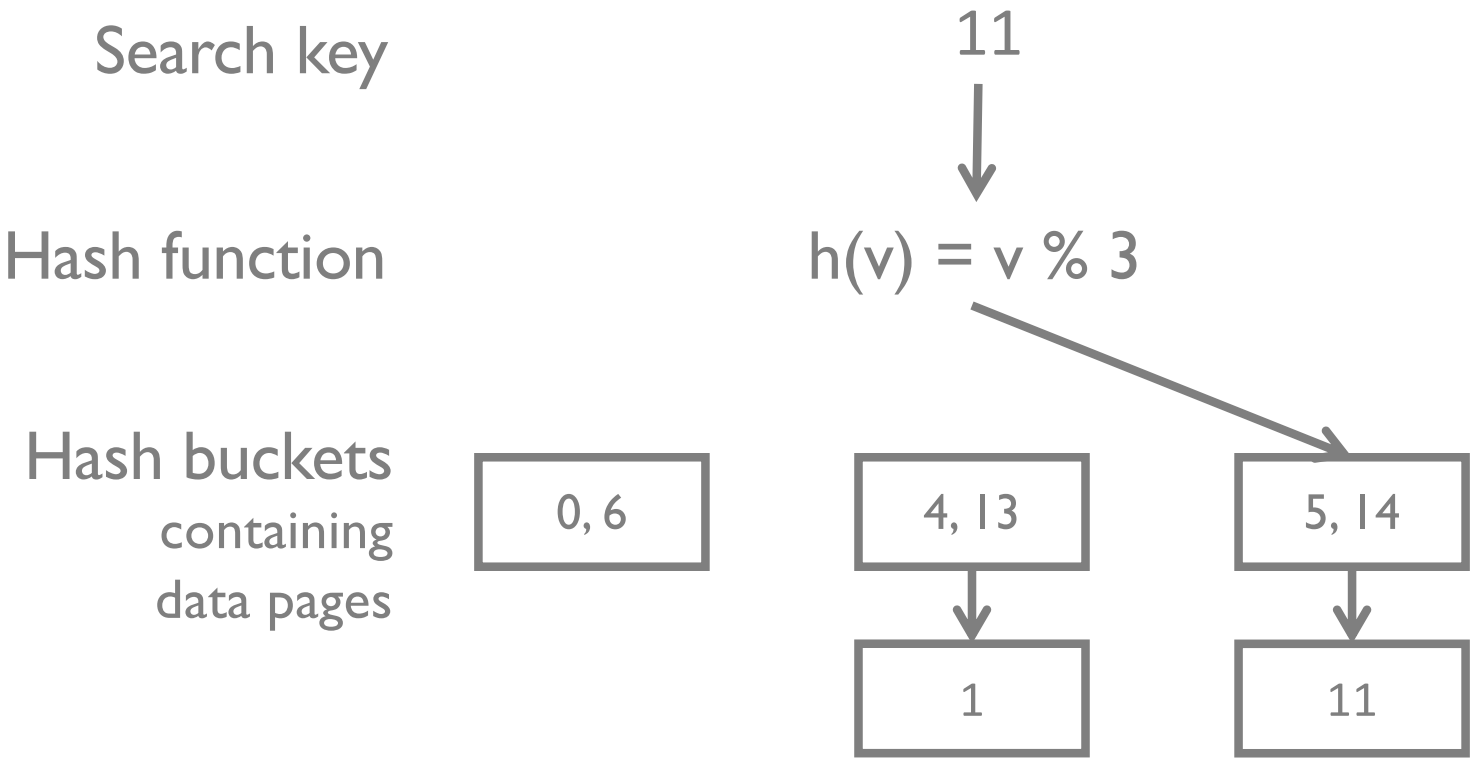
Search key 11

Hash function h(v) = v % 3

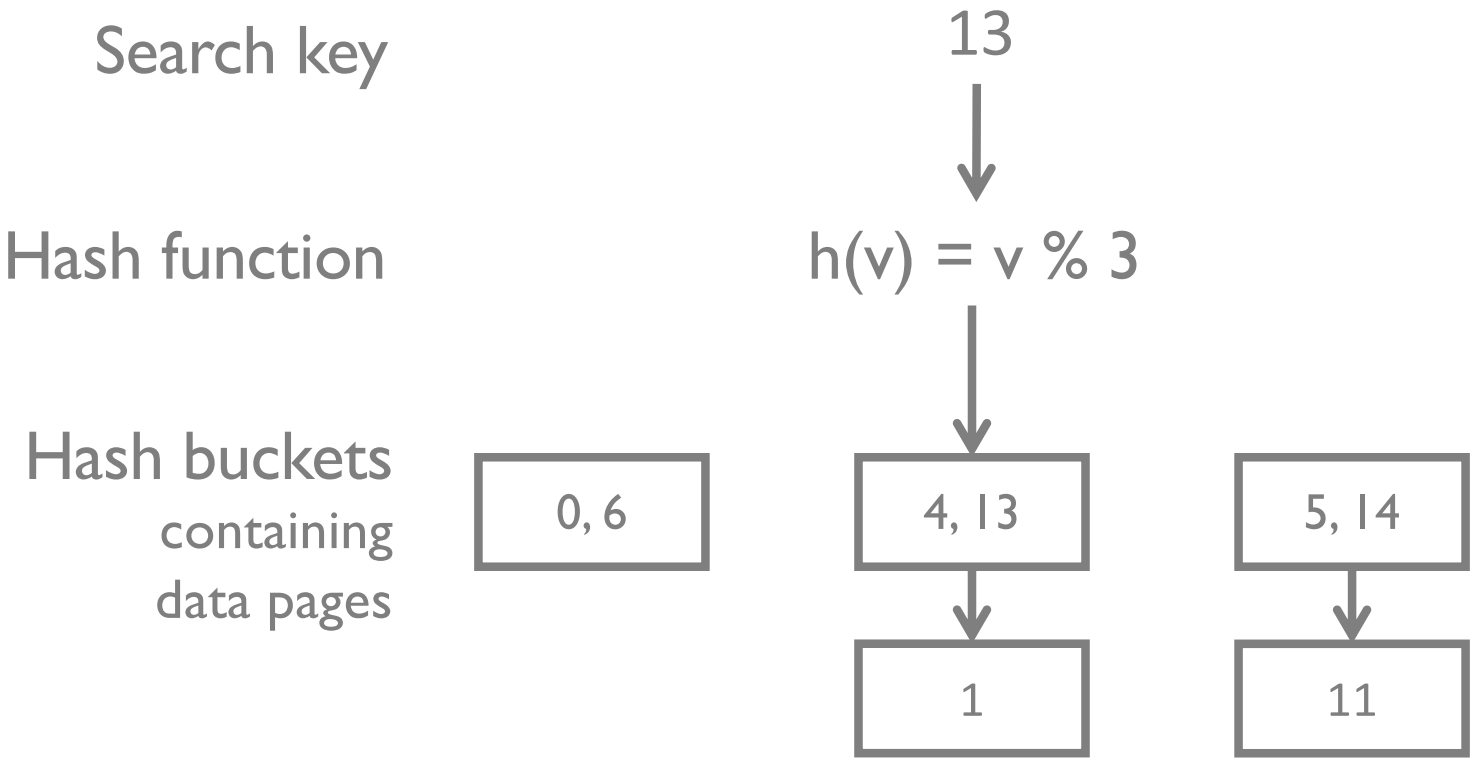
Hash buckets containing data pages



INSERT Hash Index on age



SEARCH Hash Index on age



Good for equality selections

Index = data pages + overflow data pages

Hash function h(v) takes as input the *search key*

Costs

Three file types

- Heap, B+ Tree, Hash
- Indexes can be primary or secondary

Operations we care about

- Scan all data — SELECT * FROM R
- Equality — SELECT * FROM R WHERE x = 1
- Range — SELECT * FROM R WHERE x > 10 and x < 50
- Insert tuple
- Delete tuple — DELETE WHERE …

	Heap File	Sorted Heap	B+ Tree	Hash
Scan everything				
Equality				
Range				
Insert				
Delete				

The above table's calculations will be based on assumptions about the fill factors, directory and data entry sizes, page sizes, etc. They estimate the cost in expectation.

The assumptions may vary depending on the system and configuration.

B # *data* pages

D time to read/write page

M # pages in range query

	Heap File	Sorted Heap	B+ Tree	Hash
Scan everything	BD			
Equality	0.5BD			
Range	BD			
Insert	2D			
Delete	Search + D			

Heap File

equality on a key. How many results?

B # *data* pages

D time to read/write page

M # pages in range query

	Heap File	Sorted Heap	B+ Tree	Hash
Scan everything	BD	BD		
Equality	0.5BD	$D(\log_2 B)$		
Range	BD	$D(\log_2 B + M)$		
Insert	2D	Search + BD		
Delete	Search + D	Search + BD		

Heap File

equality on a key. How many results?

Sorted File

files compacted after deletion

B # *data* pages

D time to read/write page

M # pages in range query

	Heap File	Sorted Heap	B+ Tree	Hash
Scan everything	BD	BD	1.25BD	
Equality	0.5BD	$D(\log_2 B)$	$D(\log_{80} B + 1)$	
Range	BD	$D(\log_2 B + M)$	$D(\log_{80} B + M)$	
Insert	2D	Search + BD	$D(\log_{80} B + 2)$	
Delete	Search + D	Search + BD	$D(\log_{80} B + 2)$	

Heap File

equality on a key. How many results?

Sorted File

assume: files compacted after deletion

B+ Tree

100 entries/directory page

80% fill factor

B # *data* pages

D time to read/write page

M # pages in range query

	Heap File	Sorted Heap	B+ Tree	Hash
Scan everything	BD	BD	1.25BD	1.25BD
Equality	0.5BD	$D(\log_2 B)$	$D(\log_{80} B + 1)$	D
Range	BD	$D(\log_2 B + M)$	$D(\log_{80} B + M)$	1.25BD
Insert	2D	Search + BD	$D(\log_{80} B + 2)$	2D
Delete	Search + D	Search + BD	$D(\log_{80} B + 2)$	2D

Heap File

equality on a key. How many results?

Sorted File

assume: files compacted after deletion

B+ Tree

100 entries/directory page

80% fill factor

Hash index

80% fill factor when computing scans

assumes no overflow when computing equality/insert/delete.

B # *data* pages

D time to read/write page

M # pages in range query

Administrivia

Extra credit 2 is out (2%)

- analyze Kaggle competition data using SQL
- compare analysis using normalized and denormalized schemas
- help do research (optional)

▸ Consent Form

Purpose of this study: We hope to use assignment submissions as part of a research study. The purpose of this study is to understand how easy or difficult it is to write correct analysis queries over normalized and denormalized databases. This will help us understand challenges in database decomposition, and inform the development of simpler and more correct querying interfaces. If you have any questions about the study, please reach out to Zachary Huang (zh2408@columbia.edu).

Freedom to withdraw: Your choice to participate in this study is completely voluntary and does not affect your extra credit grade. If you want to withdraw your consent in the future, please contact Zachary Huang. We will destroy any data we hold from you.

Privacy & Data Security: We will ask you to write SQL queries using normalized and denormalized databses, and fill out surveys. We do not collect any other personal information and the data are anonymous. We will carefully examine the words and exclude any details that may contain identifiable information. Identifying information involved during recruitment (uni) will not be stored for the study. We may publish research reports that include your comments and actions, but they will be anonymous.

If you are below 18, we will not use your data.

If you are over 18 and are willing to participate in the study, please answer yes. If you are below 18 or decline to participate in the study, please answer no.

`consent:` yes

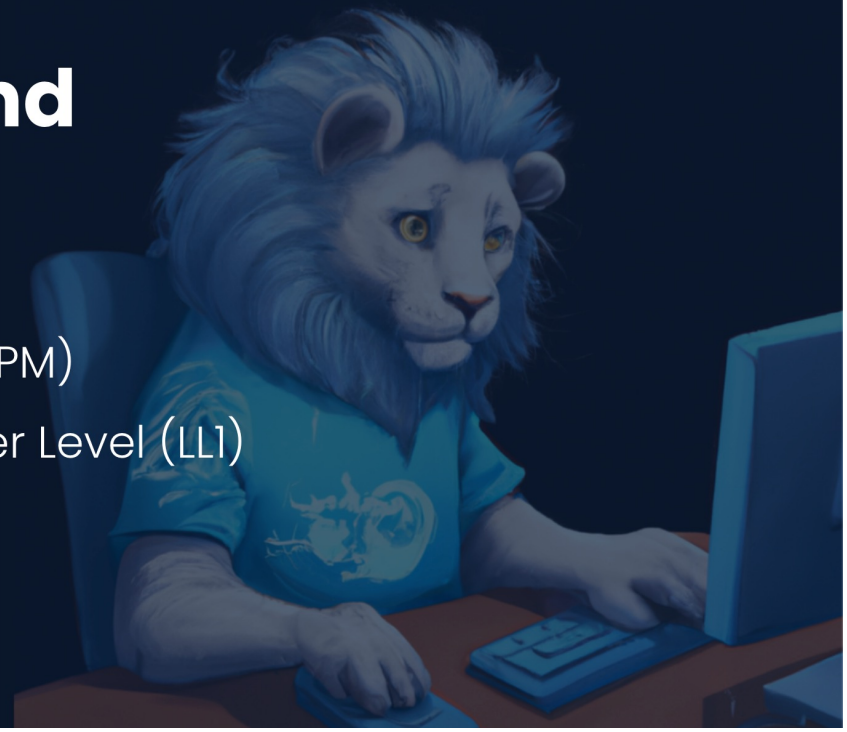
ucdsfair.com

Columbia University Data Science Institute | Vagelos Computational Science Center, Barnard College | CS@CU Computer Science

Home Sponsor Exhibitors

Register Here

Undergraduate Computer and Data Science Research Fair



Event Date: Friday, November 11, 2022 (2:00 PM - 6:00 PM)

Location: Barnard Diana Center, The Event Oval, Lower Level (LL1)

Register Here

Where do B, D, M come from?

Estimated from more basic info

Assuming (we will be clear about this)

- fanout = number of directory entries
- pointer in secondary index same size as directory entry

Given:

- p: page size
- r: record size
- d: directory entry size
- f: fill factor
- n: # records

Estimate for primary and secondary index:

- size
- height
- access cost

Primary B+ Index

Page Size	p = 100	records/page	p / r = 10
Record Size	r = 10	direntries/page	p / d = 20
Dir entry Size	d = 5	fanout:	20
Fill Factor	f = 100%	# data pages	n/(p/r) = 800
# Records	n = 8000	height	$\log_{20}800 = 3$

Cost to look up a single record is
3 for directory pages + 1 data page

Secondary B+ Index

Page Size	p = 100	records/page	p / r = 10
Record Size	r = 10	direntries/page	p / d = 20
Dir entry Size	d = 5	fanout:	20
Fill Factor	f = 100%	# data pages	n/(p/d) = 400
# Records	n = 8000	height	2

Cost to look up a single record is
2 for directory pages + 1 data page + 1 pointer lookup

How to pick?

Depends on your queries (workload)

- Which relations?
- Which attributes?
- Which types of predicates (=, <,>)
- *Selectivity*
- Insert/delete/update queries? how many?

How to choose indexes?

Considerations

- which relations should have indexes?
- on what attributes?
- how many indexes?
- what type of index (hash/tree)?

Naïve Algorithm

```
get query workload
group queries by type
for each query type in order of importance
   calculate best cost using current indexes
   if new index IDX will further reduce cost
      create IDX
```

Why not create every index?

- update queries slowed down (upkeep costs)
- takes up space

High level guidelines

Check the WHERE clauses

- attributes in WHERE are search/index keys
- equality predicate → hash or tree index
- range predicate → tree index

Multi-attribute search keys supported

- order of attributes matters for range queries
- may enable queries that don't look at data pages (*index-only*)

Summary

Design depends on economics, access cost ratios

Disk still dominant wrt cost/capacity ratio

Many physical layouts for files

- same APIs, difference performance
- remember physical independence

Indexes

- Structures to speed up read queries
- Multiple indexes possible
- Decision depends on workload

Things to Know

How a hard drive works and its major performance characteristics

The storage hierarchy & differences between RAM, SSD, Hard drives

What files, pages, and records are, and how different than UNIX model

Heap File data structure

B+ tree and Hash indexes

Performance characteristics of different file organizations

Given statistics, figure out directory size, index height, access cost